1

AN EMPIRICAL DEVELOPMENT OF PARALLELIZATION GUIDELINES FOR TIME-DRIVEN SIMULATION

THESIS

Mark Leslie Huson
Captain, USAF

AFIT/GCS/ENG/89D-10

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

1

AFIT/GCS/ENG/89D-10

AN EMPIRICAL DEVELOPMENT OF PARALLELIZATION GUIDELINES FOR TIME-DRIVEN SIMULATION

THESIS

Mark Leslie Huson
Captain, USAF

AFIT/GCS/ENG/89D-10

Approved for public release; distribution unlimited

AFIT/GCS/ENG/89D-10

AN EMPIRICAL DEVELOPMENT OF PARALLELIZATION GUIDELINES FOR TIME-DRIVEN SIMULATION

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Science)

Mark Leslie Huson, B.S., M.S.

Captain, USAF

December, 1989

Approved for public release; distribution unlimited

Acknowledgments

There are many individuals who deserve thanks for their support in this research effort. First, I would like to thank Dr. Thomas C. Hartrum, my thesis advisor, for his guidance, comments, suggestions, and ideas. Without his help and encouragement this document would not exist. I would also like to thank the members of my thesis committee, Maj William Hobart, Capt Bruce George, and CPT Robert Hammell. They certainly earned my gratitude for providing badly needed input and my respect for their patience and thoroughness in reviewing my appropriately named *rough* draft.

I would also like to thank all those people who made life and learning at AFIT bearable. The entire GCS section comes to mind for the infrequent but necessary respits from studying. In particular my fellow Cub fans, Captains Bill Harding and Mike Proicou and their wives, and Captains Steve March and Gary Whitted and their wives, who though not Cub fans, put on a good show during our trips to watch some games.

I would also like to recognize the help of my professors at the University of Tulsa. They will never know how well they prepared me for AFIT and I'm not sure they would believe me if I told them. Ellen, Suny, Cathy, Cindy and Liz at Moore's Nautilus would be equally surprised if I were to tell them how much they helped me to maintain my sanity in what otherwise would have been an impossible program.

Finally, I would like to thank my parents for their support and encouragement. No amount of effort on my part can repay the love and patient guidance they have provided over the years. It may have seemed that I paid no attention or didn't care, but they were never discouraged, and for that I am thankful.

Mark Leslie Huson

Table of Contents

Page

List of Figures

Figure Page

List of Tables

Table | Page

AFIT/GCS/ENG/89D-10

Abstract

Distributed simulation is an area of research which offers great promise for speeding up simulations. Program parallelization is usually an iterative process requiring several attempts to produce an efficient parallel implementation of a sequential program. This is due to the lack of any standards or guidelines for program parallelization.

In this research effort a Ballistic Missile Defense (BMD) time driven simulation program, developed by DESE Research and Engineering , was used as a test vehicle for investigating parallelization options for distributed and shared memory architectures. Implementations were developed to address issues of functional versus data program decomposition, computation versus communications overhead, and shared versus distributed memory architectures.

Performance data collected from each implementation was used to develop guidelines for implementing parallel versions of sequential time-driven simulations. These guidelines were based on the relative performance of the various implementations and on general observations made during the course of the research.

AN EMPIRICAL DEVELOPMENT OF PARALLELIZATION GUIDELINES FOR TIME-DRIVEN SIMULATION

I. Introduction

1.1 Why Distributed Simulation?

Distributed simulation has received much research attention in the last decade. The principal goal of distributed simulation is to improve the performance of simulations, usually with respect to time. Simulation is commonly recognized as a computationally intensive activity (5, 15, 24, 34, 35, 40). The increased availability of cheap, powerful microprocessors has resulted in commercially feasible multiprocessor computer systems, which, in turn, has increased the opportunities and incentives for development of distributed simulation methods (18). Distributing a computational process across multiple processors increases the computing power applied to a specific problem, and should reduce the "real time" needed to solve the problem. This is the essence of distributed simulation and of parallel computation in general.

According to Gilmer and Hong, "Parallel processing offers the possibility of greatly increased performance for simulations which are computationally bound on existing machines" (15:430). Computation time for many important simulations is prohibitive with even the fastest sequential computers. For example, Quinn states that a simulation to produce a 24-hour weather forecast for New York, Washington, D.C., and Philadelphia would require 24 hours to complete on a 100 megaflop sequential computer (the equivalent of a Cray-1). This type of time constraint is common to the areas of weather prediction, aerodynamics, artificial intelligence, analysis of satellite information, nuclear reactor safety, large digital logic circuits, and military simulations (35:2).

1.2 *The Simulation Process*

Biles defines simulation as "the development of a mathematical-logical model of a system and the experimental manipulation of the model on a digital computer" (5:7). Similarly, Banks and Carson define it as "the imitation of the operation of a real-world process or system over time" (1) and Shannon calls it "the process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the behavior of the system or of evaluating various strategies for the operation of the system" (40). The two basic concepts common to these definitions are to produce a model of a system, and to perform experiments using that model.

Simulation models are the mechanism through which simulation occurs. Models are designed to encapsulate the essential features of the system under study (5). Computer models must be constructed in terms of computable functions and, as such, require the adoption of a particular view or paradigm of the system. The resulting model represents the simulation view of the "real world" or at least those aspects of interest to the experimenter (34). This system model is what is translated into a computer program and implemented as a simulation.

1.2.1 Simulation Categories Simulation models fall into three general categories according to Pritsker. *Discrete simulation* models, also called discrete event models, involve dependent variables which change discretely at specified points in simulation time referred to as event times. The time variable is either continuous or discrete, depending on whether the event times can occur at any point in time or only at specified points. *Continuous simulation* models, known as time driven models, have dependent variables which change continuously over simulated time. Such models are either continuous or discrete in time, depending on whether the values of the dependent variables are available at any point in simulated time or only at specified points in simulated time. *Combined simulation* models are characterized by dependent variables which change discretely, continuously, or continuously with discrete changes superimposed. The distinguishing feature of these simulations is the interaction between discretely and continuously changing variables (34).

1.2.2 Implementation Languages Biles lists three general classes of languages for implementing simulation models. *High order languages* such as FORTRAN, C, Pascal or Ada may be used for the implementation. *General purpose languages*, such as GASP-IV, Simscript, and SLAM-II, provide more direct support for accepted simulation practices. In some instances the model implementor may benefit from using a *special purpose language*, such as GPSS, though these languages are geared primarily to a very specific area of application (5:9). High order languages are most common in current distributed simulation work because few general purpose or special purpose simulation languages are available on multiprocessor systems.

1.3 Distributed Simulation

Kaudel identifies three kinds of parallelism in simulation models which can be exploited to speed up parallel implementation of these models. Executing multiple independent trials of a simulation model is considered *application level parallelism*. Performing simulation overhead activities on separate processors, while retaining an essentially sequential simulation model, is *support function distribution*. Execution of a spatially decomposed model is *model function distribution* (27). Jones proposes an alternative approach to distributing the model based on temporally decomposing the model in a manner similar to instruction pipelining, though it could be argued this approach is an extension of Kaudel's model function distribution (26).

1.3.1 Current Research Distributed simulation models fall into the same categories as general simulation models. They can be discrete event, time driven, or a combination of the two. The majority of recent studies have addressed the category of discrete event simulation. Current research (4, 14, 23, 29, 30, 31, 36, 37) has also concentrated on the model function distribution approach to model decomposition. In all cases there is agreement that distributing a simulation across multiple processors can decrease the execution time of simulations. However, simulation model distribution has a price. Implementation is made more complex by the decisions to be made during model decomposition and by the effects these decisions have on simulation performance (2, 10, 28, 31, 40).

Distributed simulation introduces problems into the simulation model which are not present in sequential simulations (6, 7). Early distributed computing paradigms recognized the potential for deadlock in any system of communicating processes (19). Additionally, while partitioning the model among several processors increases the amount of work which can be accomplished over any period of time, the overhead incurred in distributing the model may be more than the benefits gained by partitioning the model in the first place (18).

Many model level concerns unique to distributed simulations depend on the kind of parallelism exploited in distributing a simulation model. Application level parallelism introduces few unique problems in model implementation because each independent trial, by definition, is simply an instance of a sequential model. The problems introduced are primarily resource contention problems, similar to the problems faced in operating systems. For this reason it has received little attention in the literature (27). Through his experiments, Comfort discovered support function distribution is limited by the amount of parallelism present in the support functions (accumulating statistics, managing event lists, generating pseudorandom numbers, etc.) and the portion of computation required to accomplish these functions. Comfort's results revealed that minimal speed up can be expected when applying support function distribution (11). For these reasons, model function distribution has received most of the research attention.

1.3.2 Hardware Perspectives One of the major factors making distributed simulation research feasible has been the introduction of commercial multiprocessor systems. The hardware architecture selected for the simulation implementation can have a profound effect on the efficiency of a given simulation model (2, 10, 28). There are two basic types of multiprocessor systems available. One type is composed of processing elements, each with its own local memory, tied together via an interconnection network used for passing messages. These systems are known as distributed memory systems, examples of which are the Intel iPSC family and the BBN Butterfly. The second type is shared memory systems, which are characterized by a collection of processors which access a large, usually partitioned, memory space, and whose processors communicate via the memory system. The shared memory systems have the advantage of not requiring explicit message passing and its associated transmission delays, but they are limited by the number of processors which can be in the system due to the increased memory bandwidth required to allow concurrent or shared memory accesses with minimal memory contention (20). Examples of shared memory systems are the Encore Multimax, and the Sequent Balance.

1.3.3 Research Approaches Bryant and Chandy–Misra were at the forefront in proposing approaches to solving the problems inherent in distributed simulation (6, 7). Further research has lead to the identification of approaches as either *conservative* or *optimistic*. According to Reynolds, "Algorithms are *conservative* if they satisfy the property that no process receives information from any other process that predates the current simulation time of the receiving process" (39). Some of the approaches considered conservative include deadlock detection (9), SRADS (38), appointments (32), and conditional events (8). In contrast, Reynolds identifies algorithms as *optimistic* "if processes can act on incomplete information, thus admitting the case where messages may arrive "in the past"" (39:325) (Quinn calls this *relaxation* in reference to general parallel program design (35)). The optimistic approach is based on the concept of "virtual time" proposed by Jefferson and Sowizral (23, 24).

The best known example of optimistic algorithms is the Time Warp operating system, developed by Jefferson et al. (25).

Typically, researchers view approaches to the problem of distributed simulation as falling into one of these two categories. However, Reynolds contends there is a "spectrum of options" for which these two categories only represent different portions of the spectrum. Reynolds also proposes a method of describing approaches within the spectrum and demonstrates his method by developing descriptions of some of the most commonly recognized approaches (39).

1.3.4 Implementation Concerns The selection of an approach to solving problems in distributed simulation only addresses part of the difficulty of implementation. Decomposing a system, using Kaudel's kinds of parallelism, requires careful attention to the process of mapping the implementation to an available architecture (2, 31). When properly accomplished, a simulation model is distributed across a computing environment in such a way as to minimize communication between processors while balancing the workload so all processors are performing under essentially the same computational load (18, 31, 39). Mapping model processes to physical processors is further compounded by variations between the architecture a simulation is originally targeted for and the architecture on which it may eventually be required to run (10, 28, 42).

1.4 Problem Statement

Distributed simulation, as an area of research, is still in its infancy. Nearly all work which is being done in the area is empirically based. Simulations are decomposed in an ad hoc manner to address the concerns of load balancing, process communication, selection of architecture, and overall decomposition approach. Usually, a researcher will try a decomposition and mapping for whatever architecture is available, collect statistics on the simulation performance, and accept or reject the decomposition and mapping based on the collected statistics.

One of the major problems with distributing simulations is the lack of guidelines or heuristics for the decomposition, process mapping, and architecture selection. The amount of information required and the level of effort necessary to make informed decisions for these important aspects of simulation have led to the experimental approach to decomposition, process mapping, and selection of a target architecture. For distributed simulation to become practical, it is necessary to formalize these decisions to the extent that decisions, based on an understanding of the distributed simulation process, can be made with incomplete information and with reasonable certainty of improving performance.

A formalized approach is particularly important for the process of "parallelizing" an existing simulation. Virtually all existing simulations are implicitly sequential in their design and implementation. For these simulations to take advantage of the performance offered by distributing their processing in a parallel environment, intelligent decisions must be made to decompose them into parallel processes which can then be mapped to a selected architecture.

The goal of this effort is to develop a set of guidelines or a methodology for distributing existing sequential simulations. These guidelines include methods for performing simulation decomposition, selecting an appropriate synchronization approach, and selecting an appropriate architecture for the distributed simulation.

1.5 Scope

This research effort is limited to the area of time driven simulation. Specific topics considered include methods of analyzing and decomposing simulations to represent the parallelism in the simulation, selection of functional and/or data partitioning for multiple processors, choice of an implementation approach ("optimistic", "conservative", etc.), and selection of a specific architecture for implementation of the distributed simulation. These topics are not completely independent, and the interactions between them are also addressed in this effort. The area of distributed discrete event simulation will not be addressed in this effort.

1.6 Approach

The Ballistic Missile Defense simulation, hereafter referred to as the BMD simulation or BMDSIM, was used as a test vehicle for this research effort. This simulation is a time driven battle management simulation which exhibits many of the computational characteristics of the "typical" battle management/command and control simulations used for military simulation.

The first step in this effort involves a detailed analysis of the existing sequential simulation to determine the data dependencies and relative computational loads of the functional modules of the simulation. This requires both compile and run time analysis of the source code using source code analyzers and profiling tools available under the Unix operating system.

Several methods are applied to characterizing the functional parallelism within the sequential simulation. Both verbal and graphical representations are used. Data flow diagrams and process dependency graphs of the existing functions are created to represent and understand the possible decomposition, and the dependence of these decompositions on the sequential programs data and control flow.

Once the parallelism has been expressed, the representations are used to map the independent functions to an Intel iPSC/1 Hypercube parallel architecture representing the class of distributed memory machines. Once a distributed version of the simulation is running on this architecture, modifications are made to the running simulation to evaluate the performance of the simulation with respect to the simulation approach taken, the number of processors used, and the structure of the functional decomposition and their effect on the overall performance of the simulation. A data decomposed system is implemented on the same architecture in order to compare the relative performance of alternative simulation decompositions.

The same implementation and evaluation steps are then performed for an Encore Multimax computer representing the class of shared memory architectures.

Performance results from these implementations are evaluated for execution time, efficiency, and "speed up". In addition, the ease of implementation is also addressed, but this is of necessity a subjective measurement. The performance results are finally correlated to the characteristics of the simulation and the architecture used in the program implementation.

The final result of this thesis effort is a set of guidelines based on the preceding steps. The correlated performance results and experiences during this process provide the criteria for development of the guidelines. The guidelines include "optimal" architecture and simulation approach selection based on the characterization, and methods of compensating for "non-optimal" architectures.

The approach outlined here was selected primarily because of the availability of existing simulations and hardware at the Air Force Institute of Technology. Since the BMD simulation was available, and its sequential execution exhibited the extensive computational requirements common to many of the candidate areas for distributed simulation, it seemed suitable as a test vehicle for this research. In addition, the availability of the Intel iPSC/1 Hypercube and the Encore Mu' ıax made them logical choices as representatives of their respective architectures.

1.7 Overview of the Thesis

The remaining chapters of this thesis represent the body of research developed in this effort. Chapter 2 is an analysis of the major issues in parallel simulation which affect the potential performance of a parallelized simulation. Special attention is paid to issues specific to the areas of simulation and time driven simulation. This chapter concludes with issues related to the problem of parallelizing existing programs and simulations.

Chapter 3 provides background information on the specific architectures used during this effort. Architecture details are supplemented with message passing characteristics for distributed memory systems and memory and bus information for shared memory systems.

Chapter 4 is a general processing description and program analysis for the sequential BMD simulation. This information is necessary to understand the various implementations of the simulation created during this effort.

Chapter 5 then describes each of the parallel implementations for the BMD simulation. This information includes a general description of each implementation, the rationale for the implementation, and a statement of the expected performance of the simulation.

Results for all implementations are contained in chapter 6. This chapter also includes an analysis and comparison of the results, which is then used to develop a final set of guidelines.

The conclusion, contained in chapter 7, summarizes this thesis and recommends areas for further research.

II. *Issues in Parallel Simulation*

Any discussion of parallel simulation requires an understanding of the major issues involved. This chapter outlines some of the issues to be considered both in parallel simulation and parallel programming in general, though the emphasis is on simulation.

2.1 What are the issues?

2.1.1 Feasibility When is it possible to parallelize a program? It is generally recognized that any program with a set of independent processes is a candidate for parallelization. Unfortunately, many programs exhibit complex dependency relationships which makes identification of independent processes more difficult. Misra noted "the typical simulation algorithm does not easily partition for parallel execution" (29). Most programs are composed of a set of procedures and functions which have either time or data dependencies (precedence ordering).

These dependencies can be identified by creating a dependency graph of the processes to be executed. This graph represents the time and data dependencies explicitly with arcs between nodes (where the nodes are processes and the arcs represent dependencies). Data dependencies are determined by intersecting the domain and range of each process with the ranges of other processes. Any non-empty intersection indicates a data dependency exists between the two processes. In contrast, the time dependencies are identified by the control structure of the simulation algorithm and the procedures and functions used to implement the simulation.

The data granularity used in identifying independent tasks will affect the perception of the candidate program as either feasible or infeasible. For example, on a macro scale a data structure may appear in both the range and domain of two separate processes. At this level of observation, a dependency relation exists between the two processes. If the same components of the data structure are used or modified by the processes, the dependency does exist. However, if the processes use separate components of the data structure, no dependency exists.

The overall structure of this dependency graph can have one of three general patterns. An *acyclic directed graph with multiple paths* would represent a typical discrete event simulation with no feedback. For example, a graph for a car wash simulation might be represented by Figure 2.1. In contrast a typical time driven

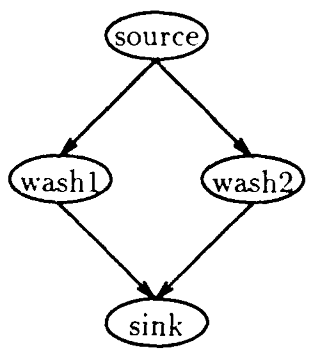

Figure 2.1. Dependency Graph for A Car Wash Simulation

simulation would be represented by a *directed graph with at least one cycle*. A simplified graph of the inner loop of the BMD simulation, Figure 2.2, is an example of this sort of graph. The final dependency graph pattern a program might have is an *acyclic single path directed graph* or pipeline. This type of graph is a good representation of an assembly line, Figure 2.3. While programs with a pipeline dependency graph may be impossible to functionally decompose, data decompositions may be possible depending on the computational dependencies between data items and any relaxation of data interdependencies.

2.1.2 Interprocess Communication For processes to obtain the data needed in a distributed processing environment, some mechanism must exist to exchange information. The hardware mechanism used is determined by the type of architecture used for program implementation. Distributed memory systems use message passing through interconnection networks as the means of exchanging information, while

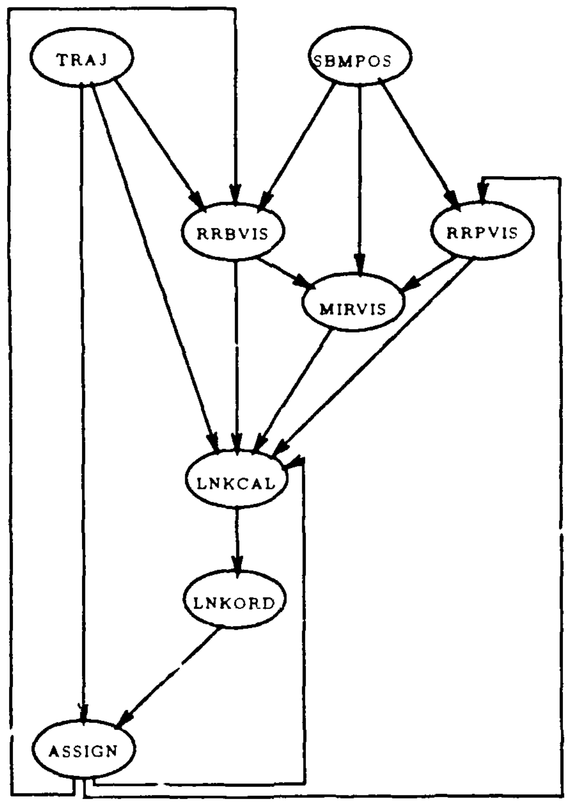

Figure 2.2. Simplified Dependency Graph of the BMD Simulation

shared memory systems can use shared data (or message passing via operating system features such as Unix pipes and sockets or other mechanisms) as their means of exchanging information.

System overhead associated with communication is determined, in part, by the mechanism used and the way it is implemented. The Intel iPSC architectures used in this effort provide examples of distributed memory (message passing) systems with and without co-processors to handle interprocess communication. Another factor affecting the communications overhead is the interconnection network of the system. In a fully connected system the overhead for communication between any two processors is not a function of which two processors are communicating. In a non-fully connected system this overhead becomes a function of the number of intermediate processors which must relay the message traffic between the two "communicating"

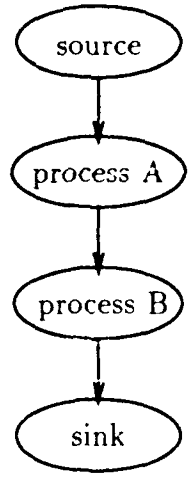

Figure 2.3. Dependency Graph of an Assembly Line

processors.

2.1.3 Synchronization In sequential simulation a single process maintains the system state and executes in the order prescribed in the simulation design. No computations can occur out of order and process synchronization is controlled by the program instruction pointer. When simulation computations are distributed across multiple processors (Multiple Instruction Multiple Data) in a parallel architecture, the instruction pointer is no longer a dependable mechanism for coordinating computational activity because each processor has its own pointer into the code it is executing.

Synchronization is necessary between processes which must share or exchange information. In distributed memory systems, the only dependable means of synchronizing computation is through message passing. Clock skew between processors makes the use of system count down timers or wall time unpredictable as a synchronization mechanism. Since the goal of parallel processing is to apply more computational resources to a problem, minimizing synchronization overhead is one of the primary activities of problem decomposition. The desired result is that each process spends more time computing rather than synchronizing with other process. Shared memory systems generally have less overhead for process communication and synchronization.

2.1.4 Deadlock Deadlock is usually defined as a state where all processes in a set of processes are blocked; and each is waiting for an event which can only be caused by another process in the set (33:275). This requires the conditions of mutual exclusion, hold and wait resources, no resource preemption, and circular wait. For simulations, deadlock refers to the situation where the simulation does not progress (simulation time does not increase).

Deadlock in discrete event simulation is primarily due to the message traffic between logical processes and the use of event times to advance the simulation clock of each logical process. Each process is expected to process events in simulation time order.

The "conservative" programming paradigm requires all processes which receive events from more than one other process to wait until each input process sends a time stamped event. This guarantees that the process receives no events from "the past". Each process keeps track of the time of the last event received from each one of its input processes. The local simulation time for each process is the minimum time of the last events received from each of its input processes. When this minimum time changes, the process updates its local simulation clock to this new minimum by processing any pending events with times less than the new simulation time. In this case, deadlock occurs when a feedback loop exists between logical processes in the simulation. A similar situation occurs when a process never receives an event from one of its input processes. While the simulation may be correctly simulating the system and no events should be generated by the specific input process, the logical processes "down stream" from the process will not be able to advance their simulation time. Current research in "conservative" methods of deadlock avoidance concentrates on variants of the Chandy-Misra Algorithm (7).

The "optimistic" paradigm avoids deadlock by allowing each process to proceed based on the basis of its current event queue. If an event from the "past" arrives, a process will "roll back" to the simulation time in the past when the event should have arrived. Once the roll back is accomplished, the process continues processing events from the new simulation time. "Optimistic" methods require large amounts of data to be saved to allow each process to roll back to the "past". In addition, the same events may be processed several times by a logical process due to roll backs. Among the "optimistic" methods currently being researched is Jefferson's "time warp" system, which creates anti-messages during process roll back to "undo" processing which should not have been done based on the just-arrived message (25).

Deadlock in time driven simulations is usually the result of an incorrect program design. The global simulation clock ensures progress. Each process depends on the concept that all simulation processes are at the same simulation time so messages and data from the "past" cannot be sent by other processes. An improperly designed simulation allows the global simulation time to be incremented before some process has finished its processing for the previous simulation time.

2.1.5 Load Balance Ideally, a parallel program will be distributed so each processor will have the same computational load. Assuming no serial dependencies, increases in overhead, or improvements in efficiency for a parallelized algorithm, we would expect a speed up of N from a system with N processors with perfect load balance. The uniformity and consistency of the computations to be performed will often determine whether it is possible to decompose a simulation to equitably distribute the load. In the worst case, a single process will perform nearly all computations, resulting in that process limiting the speed up attainable. If the longest running process distributed in the simulation performs $\frac{1}{f}$ of the processing in the sequential simulation, the maximum speed up for the parallel simulation will be f.

Load balance is not a static condition for many computations. This complicates the process of partitioning the program among the available processors. The programmer must decide whether to statically partition the problem or to attempt to perform dynamic load balancing. Static load balancing simplifies the problem by assuming that some average load balance will provide a suitable speed up. The alternative is to periodically rebalance the computational load. This dynamic rebalancing introduces additional overhead in determining both when to rebalance, and how to repartition the program.

2.1.6 Determinism Sequential programs are deterministic. For a given sequence of input data, output results are identical for any number of trials executed. When a program is distributed, the instruction pointer no longer provides the control needed to ensure consistent results between trials. Whenever computations or data items capable of influencing program output can be processed in a random order, a program is no longer deterministic. When designing a parallel program or parallelizing an existing program, it is necessary to determine the importance of a deterministic output. From a testing standpoint, a deterministic program provides an easier platform for determining whether a program or simulation is valid.

In many simulations determinism is necessary, because the systems being simulated are deterministic systems. For such simulations program design is complicated when using a data decomposition to distribute the simulation, because data may be received in a non-deterministic order from replicated processes. Adding determinism to a non-deterministic simulation may add to a distributed simulation's computational, synchronization, or space requirements depending on the mechanism used to provide the required determinism.

2.2 Conservative, Optimistic, or a Middle Ground

Parallel computations are generally thought of as being either conservative or optimistic, based on the "quality" of information used for any computation and the mechanism used to compute the correct result. Conservative computations do not proceed until the data to be used in the computation is guaranteed to be correct. No incorrect values are generated at any point in the computation. For parallel computations this means all processes must wait until all input data is correct before proceeding. In contrast, optimistic computations continually execute based on the "best" available information at the time of computation. Since inputs are not guaranteed to be correct before computation begins, system state checkpoints are maintained to permit rollback to known correct states when erroneous processing is detected. Optimistic processes perform their computations at their own pace, without waiting for other processes; however, potential rollbacks may mean a process will perform the same computation a number of times.

As Reynolds suggested, a "spectrum" of possibilities exist between these two extremes (39). Since feedback loops tend to reduce the inherent parallelism in a simulation, a method of reducing the number of these loops or their frequency of traversal could increase the options for parallelization. Feedback loops and the information they contain can be implicit in a sequential system. A common global memory will have only one possible value for any given data item. In time driven simulations the simulation state is often maintained between time steps by data items in a global memory. In computations where the new simulation state is a function of the old simulation state, these state variables provide implicit feedback information for each ΔT.

However, some simulations contain nested loops where the values of these global variables are modified within each time interval. The program state at the end of the time interval may depend on the transitions of the state variable within these nested loops. When a simulation is parallelized on a distributed memory system, interprocess communication becomes the only method of making sure data item "A" on processor 1 is the same as data item "A" on processor 2. If feedback loops are eliminated or reduced in frequency during program parallelization, what happens if the "A" used by processor 1 is not the "current A" for the simulation? Is it possible to perform computations based on imperfect knowledge without requiring rollback? What effect will these computations have on the simulation results?

This becomes an important issue when dealing with a data decomposed system where replicated processes send intermediate results to a single non-replicated process. If the single non-replicated process changes a "global" variable this is equivalent to a critical section of a parallel program. If the non-deterministic completion order of the replicated processes can effect the results of the simulation, what can be done? Among the several options which exist are the following:

- Provide a synchronization mechanism to generate intermediate results in a deterministic order.
- Accumulate the results on the non-replicated process, waiting until all replicated processes have terminated, and process in deterministic order.
- Do not replicate the process computing intermediate results.
- Accept the non-deterministic output and correct based on some stored state space in a process later in the computation.
- Accept the intermediate results as correct and continue processing. (This will only apply in a case were determinism is not a necessary condition for the simulation.)
- Determine an acceptance criteria for the intermediate results and accept only those which meet the criteria.

In some cases eliminating feedback results only in unnecessary computations being performed. In others, it may drastically affect the results of computations. While elimination of these loops offers the potential of removing a synchronization requirement from a program, the purpose of the loop and its effect on computations must be clearly understood.

2.3 Time Driven vs. Event Driven

The major difference between time driven and event driven simulation is the mechanism for updating each process's simulation time, and the predictability of interprocess communication. Since time driven simulations operate in lock step, the communication between processes occurs in simulation time order based on the global simulation time. Each process in a discrete event simulation maintains its own simulation time. Communication in this type of environment is less predictable because messages may arrive in the past, or a process may wait for a message which never arrives before incrementing its local simulation time (a form of deadlock). Discrete event simulations eliminate the synchronization required by a global simulation clock, and attempt to speed up simulation execution by not simulating time intervals where nothing happens. The cost of this improved efficiency is that the resulting simulation may deadlock.

An advantage the synchronization of time driven simulations provides is that it reduces the possibility of program deadlock. Therefore, the mechanisms used to prevent deadlocks in an discrete event simulation are not present to add to the overhead of the simulation (specifically the overhead of null messages, deadlock detection, deadlock recovery and other techniques of "conservative" methods, or the overhead of checkpoint storage and process rollback associated with "optimistic" methods).

2.4 Parallelizing Existing Simulations

2.4.1 Deciding to Parallelize The purpose of a simulation is probably the greatest single factor to be considered in the decision to parallelize an existing simulation. A simulation which, for whatever reason, *must* take less time to execute is a candidate for parallelization. This is especially important in time critical simulations such as weather forecasting, or real time or interactive simulations such as aircraft simulators. Unfortunately, while "parallel" compilers exist, they are capable only of recognizing parallelism inherent in the source code (primarily looping constructs), and therefore depend on the programmer's ability to incorporate these structures into the code. Monolithic data structures (those not indexed by the loop variable) within loop constructs will usually defeat automated attempts at parallelization. When the target system is a distributed memory architecture, parallel compilers will most likely be unavailable to assist in distributing the simulation.

2.4.2 Level of Effort The decision to parallelize requires that the cost of the effort be recognized. The effort depends on condition of the existing simulation, its complexity, its level of documentation, and the suitability of its algorithms for parallelization. Another factor influencing the effort required is the familiarity of the programmer(s) with the simulation.

According to Glover:

> The single, most important, overriding tradeoff issue to be considered is one of efficiency. Does the programmer rewrite large amounts of the program to obtain a large speed up factor, or does the programmer rewrite some of the program to obtain a modest speedup [sic] factor? (16:1)

One obvious goal in parallelizing an existing simulation is to reuse as much of the sequential code as possible when parallelizing. The two primary advantages to this are a reduced level of effort and a presumed level of confidence in the validity of the existing code. Another issue which can add to the complexity and level of effort is language compatibility. If the language of the original simulation is not supported on the selected parallel architecture, the entire program must be converted. In addition, language extensions used in the sequential simulation may not be supported on the parallel architecture, adding to the work needed to parallelize the program.

In any event, the level of effort expended in parallelizing a simulation should be commensurate with the expected benefits of that parallelization.

III. Parallel Hardware Architectures

This chapter describes the architectures used in this research. Message passing mechanisms are explained for each of the distributed memory architectures because the mechanism used is one of the major factors effecting overall system performance. Memory locks are described for the shared memory architecture because they are most likely to cause similar performance problems for access to critical sections of shared memory.

3.1 The Intel iPSC/1

The Intel iPSC/1 Hypercube is a distributed memory architecture system. It consists of a host processor and up to 128 processor nodes configured in a hypercube topology. In a hypercube, each one of the n processing nodes has a direct connection to $log_2 n$ other nodes, and these connections are determined by the node identifier or address. Each node 0 to $(n-1)$ is connected to all nodes whose addresses differ in one bit position when expressed in binary. Figure 3.1 is a representation of a 16 node hypercube topology. All communication between the nodes in the iPSC/1 is via connected nodes, and if non-connected nodes exchange information, it must pass through and be processed by (at least for routing purposes) intermediate nodes. This means communication overhead is a function of the number of intermediate nodes between communicating processes.

Each processing node in the iPSC/1 is an Intel 80286-based processor. Node processors are configured with Intel 80287 math co-processors, 512K bytes of RAM, and eight bidirectional communication channels, managed by dedicated Intel 82586 communication co-processors (seven channels for point to point node communications and one for a shared Ethernet channel to the host processor). Message size in the iPSC/1 is limited to 16K bytes, and the total number of 1K byte blocks in transit (sent but not yet received by the application program) is limited by the

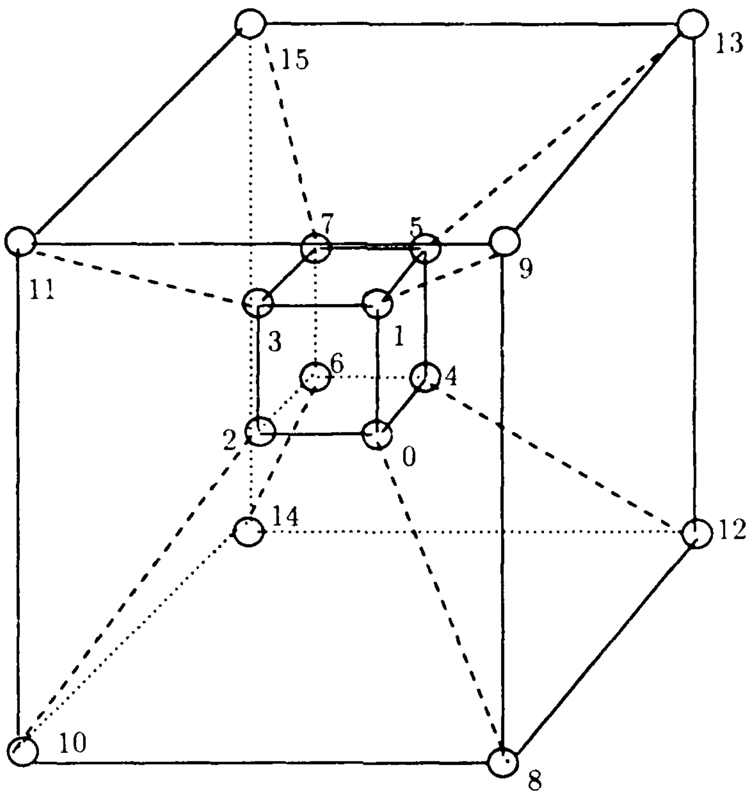

Figure 3.1. Interconnections in a 16 node hypercube

hardware configuration of the particular system. Messages larger than 1K byte are automatically broken apart, sent, and reassembled at the destination node.

Node to node communication can be viewed as two distinct processes: sending and receiving messages. Sending a message from a node application process involves the following steps:

- The CPU determines the "next node" for routing the message
- The CPU directs the appropriate 82586 LAN controller to start sending the message
- The LAN controller reads the message out of RAM
- The LAN controller sends the message out on the point to point serial link

- The LAN controller initiates an interrupt to the CPU to report that the message was sent

When a message is received at a node the following processing steps occur:

- A LAN controller receives a message from another node
- The LAN controller writes the message into RAM
- The LAN controller interrupts the CPU to report the receipt of a message and provides the address of the message to the CPU
- The CPU checks to see if it is the destination node, if not then the message must be sent using the steps outlined above
- If this is the destination, the CPU checks all processes running on the node to see if a process has a pending receive request matching the received message "type". If such a process is found, the message is transferred from a system buffer to that process's receive buffer. If no receive is pending or the type does not match, the message remains in the system buffer.

Message routing can be represented by Figure 3.2. Names on the left side of the diagram represent names of the appropriate protocol layers from the seven layer International Standards Organization's (ISO) Reference Model of Open Systems Interconnection (OSI), while names on the right indicate node component involved in that layer. (The seventh layer, the presentation layer, is not needed in internode communication.)

The host processor is also an Intel 80286-based machine with an 80287 math co-processor, and it controls the configuration and operation of cube processors. Each node communicates with the host processor via a shared Ethernet channel, controlled at the host end by an Intel iSBC 186/51 communication board. All I/O between the node processors and the external environment (i.e., printer, screen, and disk I/O) is provided by the host processor.

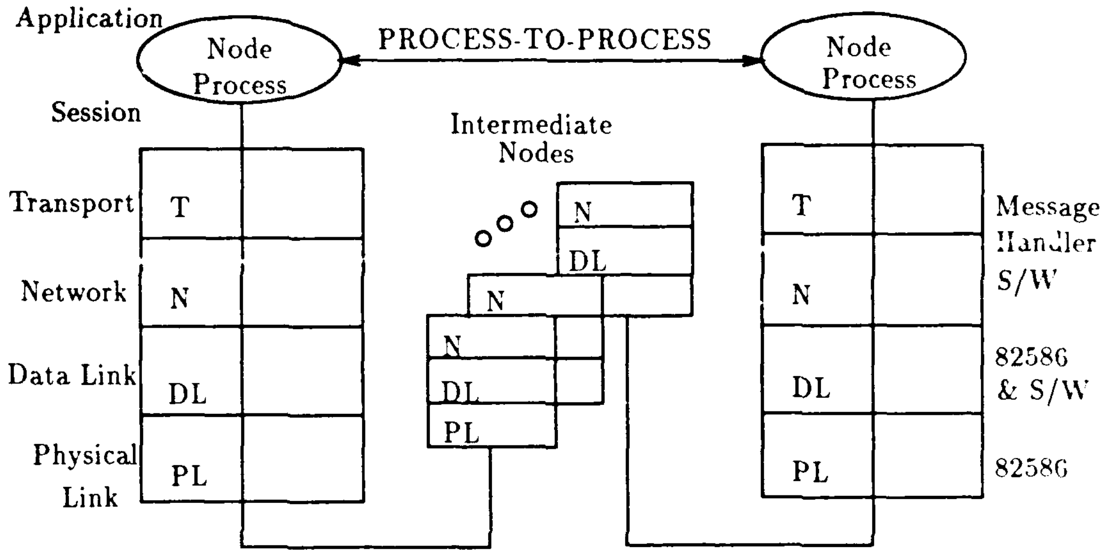

Figure 3.2. Message Routing in the iPSC/1 (21:3-5)

3.1.1 AFIT iPSC/1 Configurations AFIT has two iPSC/1 systems. One system is designated as an iPSC/D5VX, and is a 5 dimension (32 node) hypercube equipped with optional vector processor boards (one per node). The other system is an iPSC/D5MX, which is a 5 dimension hypercube with an optional 4 Megabyte memory board for each node.

3.2 The Intel iPSC/2

The iPSC/2, an 80386-based system, represents Intel's second generation hypercube architecture. While processor interconnections remain in a hypercube topology, the host and node processor configurations and capabilities have changed. Of particular interest is the mechanism for sending messages between nodes. Each node processor contains a "Direct Connect Module" which

> allows a message to be passed directly from any node processor to any other node processor, passing through only the communications modules without having to pass through intermediate node processors. This is done by a logic switching arrangement. (22:2-16)

More time is spent setting up a message for the Direct Connect module then is spent routing messages through the intermediate nodes' Direct Connect modules. This results in near uniform message latency between all nodes in the iPSC/2 whether they are physically connected or not. Node processors are involved only when they are the source or destination of a message, which increases the available time to process the user application.

3.2.1 AFIT iPSC/2 Configuration The AFIT iPSC is a 3 dimension (8 node) hypercube. While Intel offers optional memory and vector boards as well as a "disk farm" for additional mass storage, the AFIT system is not configured with any of these options.

3.3 The Encore Multimax

The Encore Multimax is a fully connected, shared memory architecture, composed of the following system components:

- Main system bus, called the Nanobus
- A system controller
- Processor cards (1 to 10)
- I/O channel cards (1 to (11 minus number of processor cards))
- Mass storage cards (1 to 8) of 4 or 16 megabyte capacity

Component interconnections are shown in Figure 3.3. System components are classified as *requesters* if they request use of the address bus but do not *respond* to requests for data. Processors cards and I/O channel cards are all *requesters*. Memory cards, which do not issue requests for the address bus but respond to requests for data, are classified as *responders*. The system controller acts as both a *requester* and a *responder*. I/O channel cards provide the Encore Multimax with access to Ethernets

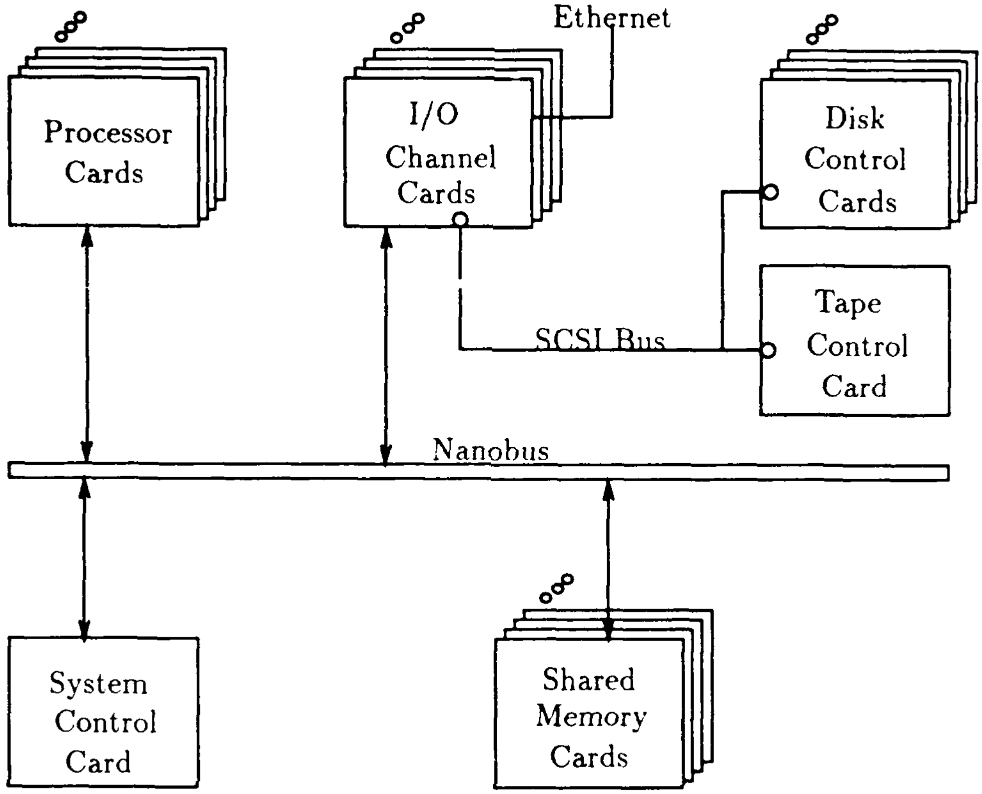

Figure 3.3. Encore Multimax Functional Diagram (13:2–4)

and mass storage devices. The other components are discussed individually in the following paragraphs.

The Nanobus is a fast bipolar bus, which provides a data transfer rate of 100 megabytes per second. This bus provides up to 12.5 million bus "transactions" per second, separate parity-protected address and data busses, a separate 14 bit wide vector interrupt bus, a separate parity-protected control bus, bus transaction interleaving, pipelined bus interfaces, and processor-memory interlocked operations (13:2–3)

The system controller performs the following functions (13:2-6):

- Supervises hardware fault diagnosis
- Performs environmental monitoring (power supplies and temperature)
- Provides interface to front panel switches and indicators

- Provides local and remote console terminal interface
- Mediates bus arbitration
- Generates bus timing signals
- Provides interval timing and time-of-year clock
- Controls system start-up, builds a configuration map of existing system resources, sizes memory, and assigns optimum interleaving characteristics

Processor cards are comprised of two independent National Semiconductor 15 MHz NS32332 processors, each with a private 64K bytes cache memory, an NS32382 15 MHz Memory Management Unit for 32-bit physical address generation, and a floating point accelerator unit using a Weitek WTK1164 multiplier and a WTK1165 Arithmetic Logic Unit.

Each shared memory card provides 4 or 16 megabytes of random access memory in two independent banks. Every card supports 2-way interleaving between banks and 4-way interleaving between cards, permitting 8-way system interleaving. The base address and interleaving characteristics of each card are set under software control at system startup. Any byte in memory can be used as a multiprocessor "lock". Atomic Nanobus operations provide the ability to set or reset the locks. A processor testing the state of a lock reads the contents into its cache, and subsequent reads are from the cache, until the value of the lock changes. As a result the Nanobus and memory card are not loaded by processes waiting for a lock to change state (13:2-12).

3.3.1 AFIT Encore Configuration The AFIT Encore system has 8 processor boards for a total of 16 processors. The system also has 32 Megabytes of main memory and one I/O channel card.

IV. The Ballistic Missile Defense (BMD) Simulation

This chapter describes the simulation used as a test vehicle for this research. A general description of the programs operation is followed by an analysis of the sequential program.

4.1 Use as a Test Vehicle

The BMD simulation, developed by DESE Research and Engineering, Incorporated, was used as the test vehicle for all applications programs developed in this research. This simulation was developed as a research task sponsored by the Defense Advanced Research Projects Agency (DARPA) under ARPA Order 3643 (12). The program exhibits characteristics which make functional, data, or functional/data hybrid decompositions possible.

4.2 Introduction and Description

DESE Research and Engineering developed two basic simulations. The first simulation (designated BMDSIM-P) generates detailed numerical data for defining optimal physical parameters and performance requirements for Directed Energy Weapon (DEW) systems. The second simulation (designated BMDSIM-G) provides a graphical interface to display attack scenario simulation results (12:2-2). The BMD simulation which forms the basis for all programs developed in this effort was a preliminary version of these simulations. It simulates a ballistic missile attack and the subsequent engagement by directed energy weapons. The term BMDSIM will be used for all subsequent references to this baseline simulation and to generic attributes of all derived simulations.

The original simulation was designed with the following guidelines and constraints:

- DEW systems were limited to concepts employing ground-based lasers and space-based relay mirrors
- BMD models only boost-phase defense engagements against strategic ballistic missiles

A booster engagement occurs in BMDSIM when the following conditions are met:

1. A booster is in boost phase and above a minimum engagement altitude
2. A geometrically feasible laser–to–booster link exists involving either one or two space-based mirrors
3. All defense elements in the feasible link are available for engagement
4. Sufficient time remains before booster burnout to complete the engagement (based on time to position defensive elements and time required to destroy the booster)

If any of these four conditions are not met, the booster is not engaged. Figure 4.1 depicts the geometry of an engagement.

The time to start an engagement is determined by the time to orient the laser and mirrors towards the booster. This time is the result of the orientations at the end of the last engagement for each defensive element, and the individual slew rates for the angles between the last engagement and the new engagement. The engagement duration is determined by the distances between objects (RRPM, RRM, and RRBM), the orientation of the laser (RPANG, divergence from vertical orientation increases atmospheric attenuation of energy), and the incident angle of the beam as it strikes the booster (RIANG). These are the major parameters used to determine the engagement time required to destroy the booster. For a more detailed discussion refer to (12).

The overall functional structure of BMDSIM is represented in Figure 4.2. Table 4.1 equates these function descriptions to the FORTRAN function names used

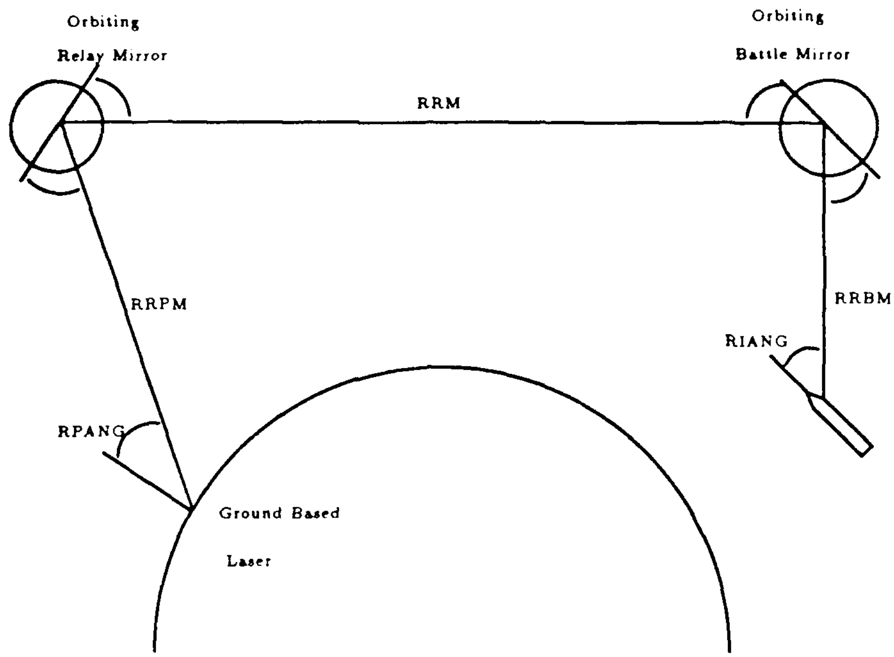

Figure 4.1. Ballistic Missile Defense Simulation Engagement Parameters (17)

in subsequent data flow diagrams and partitioning diagrams. This same structure showing the equivalent FORTRAN function names is shown in Figure 4.3.

BMDSIM models the threat missiles in terms of "centerline" trajectories representing threat "tubes" or clusters of ballistic missiles launched from a given launch complex to a specified target area. It is assumed that all numbers and types of ballistic missiles entered as threat data are modeled by their centerline trajectories. The boost phase trajectory is based on curve fits of detailed ICBM and SLBM trajectory simulation data. Overall trajectory and orbital modeling after powered flight is based on Keplerian equations. For further discussion, the reader is referred to pages 4-5 to 4-15 of (12).

Laser sites are modeled as fixed installations. All lasers are of equal power (beam intensity), and project the maximum power to the first relay mirror when it is oriented perpendicular to the surface of the earth.

Mirrors are generated by BMDSIM based on the following parameters:

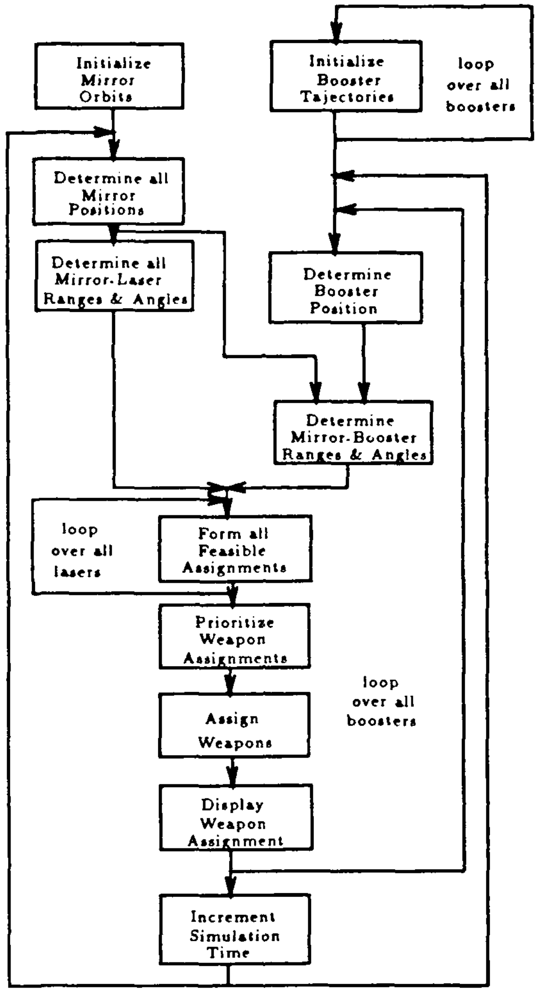

Figure 4.2. Functional Structure of BMDSIM

- The number of mirror orbits
- The number of mirrors in each orbit
- The radius of mirror orbits measured from the center of the Earth in kilometers
- The true anomaly offset between mirrors in adjacent orbits in radians
- The initial right ascension, in radians, of the first mirror in the first orbit measured counter clockwise from the Greenwich Meridian

If the true anomaly and initial right ascension are not provided, they are randomly generated. All mirror orbits are modeled as circular, geocentric orbits of unifor. ı alti-

Table 4.1. FORTRAN Function - Logical Function Equivalence

SBMIT	Initialize mirror orbits
BOSTIT	Initialize trajectory for a single booster
SBMPOS	Determine all mirror positions for a given time
RRPVIS	Determine visibility, ranges and angles for all mirrors and a single laser
TRAJ	Determine booster position and velocity vectors
RRBVIS	Determine visibility, ranges and angles for all mirrors and a single booster
MIRVIS	Determine all geometrically feasible links for a laser booster pair, by matching laser-mirror visibility, booster-visibility, and determining visibility and ranges between mirrors
LNKCAL	Calculate the time of engagement start and duration for all feasible links for a laser booster pair
LNKORD	Sort the set of feasible links for a given booster and all lasers based on time of completion for an engagement
ASSIGN	Assign weapons for the best link where all defensive elements are available for the engagement (not previously assigned)

tude. Visibility between mirrors and lasers is determined by zenith angle constraints (RPANG in Figure 4.1), and calculations are completed for all mirrors which are in the laser's hemisphere and not currently engaged against a booster. Mirror-booster visibility adds an additional constraint; if the incident angle (RIANG in Figure 4.1) is less than a predefined minimum, the link is not considered.

4.3 Analysis of the Sequential Simulation

The sequential BMDSIM was analyzed for potential parallelism and execution performance. In keeping with the idea of reusing as much code as possible in parallelizing an existing simulation, high level subroutines are viewed as the smallest possible functional decomposition units. The modular approach DESE Research and Engineering took in designing the source simulation made this view of simulation functions possible. While reducing the set of decompositions considered, this system

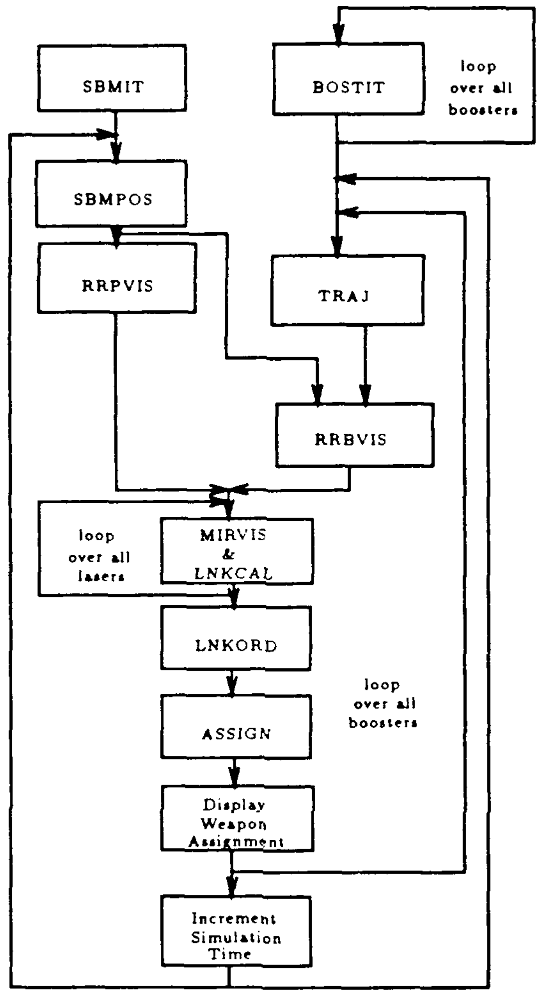

Figure 4.3. Functional Structure of BMDSIM with FORTRAN names

view simplified the task of identifying data dependencies and needed control structures. The primary devices used for high level analysis of this program were the data flow diagram in Figure 4.4 and the calling tree of the original program shown in Figure 4.5. Data structures represented in Figure 4.4 are defined in Appendix B.

A graphical display function was added to BMDSIM in a previous effort at AFIT. This function allowed simulation results to be displayed on a color Sun 3 workstation. This function was excluded from the sequential analysis. Though an Ethernet interface package exists to allow a Sun workstation to act as the host

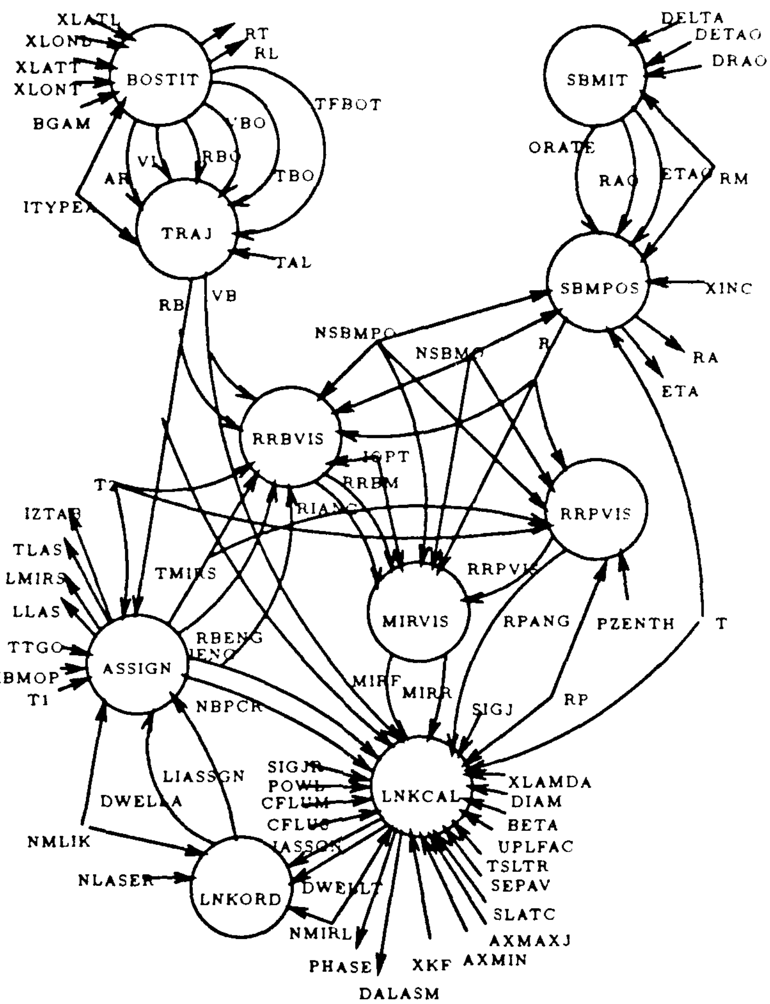

Figure 4.4. Data Flow Diagram of Sequential BMDSIM

processor for the iPSC/1, no similar package exists for the iPSC/2 or the Encore. All analysis emphasized the non-graphical versions for consistency between architectures.

The Unix profiling capability was used to determine the relative amount of time spent in each subroutine. Profile results are summarized in Table 4.2. Program profiling is a useful tool for determining the relative computational load for program subroutines. Once determined, this information can be used to indicate a general approach to take in parallelizing a program. Due to the consumption of resources

```
Start of simulation
    read in booster, mirror, laser, and control info

    call BOSTIT - initialize all booster clusters

    call SBMIT - initialize space based mirrors

    Begin simulation time loop

        SBMPOS - determine new mirror positions for
            time T

        RRPVIS - determine visibility between lasers and
            mirrors at time T

        Begin loop over booster (clusters)

            TRAJ - determine new cluster position vector

            RRBVIS - determine visibility between
                clusters and mirrors at time T

            Begin loop over lasers

                MIRVIS - determine geometrically
                    feasible laser links

                LNKCAL - determine the best "NMIRL" links
                    for this cluster from this laser

            LNKORD - sort the best "NMLIK" links for this
                cluster from all lasers

            ASSIGN - select the best laser link
                to assign to this cluster based on
                min engagement completion time

        Output information and increment time
```

Figure 4.5. Top Level Call Tree for Sequential BMDSIM

and destruction of booster clusters in BMDSIM, the computational load varies as simulation time progresses. Since the profiling information indicates only an average, performance predictions based on the results are estimates only. The accuracy of performance predictions for any implementation will also be a function of whether or not the implementation is deterministic.

Figure 4.6 is a simplified data flow diagram for the BMD simulation FORTRAN functions. This graph does not represent all of the information which the main driver program uses for control flow during the simulation. Simulation constants and some of the time step variables have been removed to indicate the major data structures

Table 4.2. FORTRAN Function - Unix Profile results

BMDSIM Profile Results		
Function	% Execution Time	Avg ms per call
main	9.7	—
MIRVIS	51.4	37.24
LNKORD	20.9	15.49
RRBVIS	9.0	6.50
LNKCAL	5.8	4.76
RRPVIS	1.5	0.84
ASSIGN	0.6	0.20
TRAJ	0.5	0.22
SBMPOS	0.3	0.10
BOSTIT	0.2	0.07
SBMIT	0.1	0.03

required by each function.

The computational load in MIRVIS is the result of an exhaustive search for feasible links. Using data item names from the source code, the search has a computational complexity of, O(*NBOSTR*NLASER*((NSBMO*NSBMPO)**2)*). Since the predominant loop within each time step is over each booster, if NBOSTR processors were used, this search would still be O(*NLASER*((NSBMO*NSBMPO)**2)*). Clearly this routine will limit the performance of any parallelization of the sequential simulation.

The logical question at this point is, based on the analysis of the sequential simulation, why not decompose or parallelize MIRVIS? Decomposing this single process was more complicated than was warranted by the expected benefits. The primary result of a decomposition/parallelization of this process would be improved performance in a functional decomposition. However, since any decomposition aimed solely at this function would still have been in a loop construct nested within both a simulation time loop and a loop over booster clusters, only a marginal benefit could

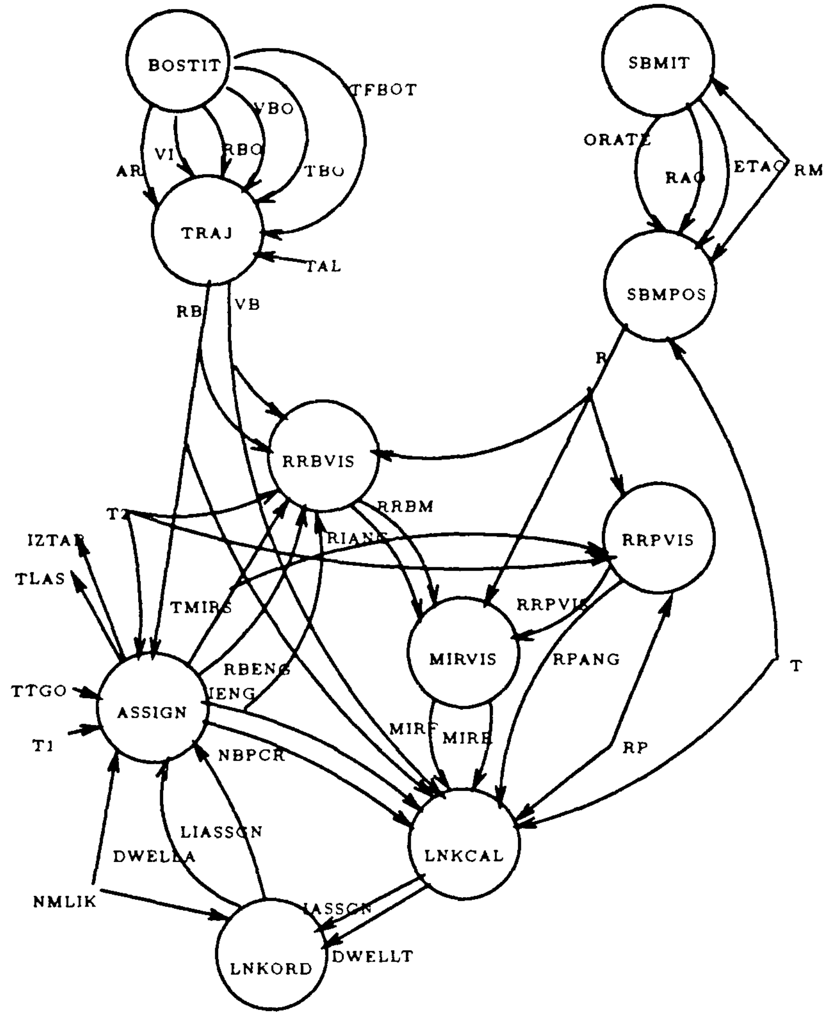

Figure 4.6. Simplified Data Flow Diagram of BMDSIM without constants

be achieved by rewriting this procedure alone.

V. Implementations of the BMD Simulation

The general strategy used in developing the implementations of BMDSIM was to begin by creating a strict functional decomposition of the original simulation for the iPSC/1. Once this first implementation existed, subsequent implementations on the iPSC/1 were created to represent a progression from functional to data decompositions of BMDSIM. The "best" implementation was then ported to the iPSC/2 system. The original intent was to also port this "best" implementation to the Encore. However, a separate Encore implementation was created to take advantage of the programming environment available on the Encore. The program analysis results from the previous chapter, including profiling results, data flow diagrams, and data structure analysis listed in Appendix B were used in developing all parallel BMDSIM implementations.

For all the implementations of BMDSIM, the functions and subroutines created by DESE Research and Engineering were left intact. While this limited the possible number of decompositions, it simplified the programming task. The trade off for this decision is the maximum obtainable speed up. Speed up is commonly defined as

$$\text{Speed up} = \frac{\text{Time for Sequential Execution}}{\text{Time for Parallel Execution}}$$

Since approximately 50% of sequential execution time is spent in one of these subroutines, speed up for a purely functional decomposition must be less than or equal to two. For a data decomposition the maximum speed up is more difficult to determine.

A speed up limit can be estimated for any implementation based on the equation

$$\begin{pmatrix} \textit{Speed up} \\ \textit{Limit} \end{pmatrix} = \left(\frac{1}{f_s + \max f_i}\right) \quad \textit{for} \quad i = 1, \ldots, N \tag{5.1}$$

where f_s is the fraction of total computation which is sequential, f_i is the fraction of the parallelized computation performed on processor i, and N is the number of processors. This equation is derived from Ahmdahl's Law. The sequential fraction of computation is based on the percent of computation performed by the "main" procedure in the sequential simulation. The fractional values used in determining the speedup limit depend on the accuracy of the Unix profiling tool. To determine the fraction of computation for a replicated process requires the equation

$$f_i = \left(\frac{S_i}{R_i}\right) \quad for \quad i = 1, \ldots, p \tag{5.2}$$

where S_i is the fraction of time spent in the process during sequential execution, R_i is the number of processors allocated to replicated process i, and p is the number of processes. However, it must be noted this limit computation completely ignores any overhead associated with process parallelization. Communication overhead for any implementation usually increases as a function of the number of processors utilized for the simulation. This is especially true in a system like the iPSC/1, where messages between non-adjacent nodes interrupts the processing on intermediate nodes. In addition, load balance for a replicated process is assumed to be perfect, and this is seldom the case for MIMD systems. Additionally, there are two reasons why $\frac{S_i}{R_i}$ is not realistic. First, a certain percentage of process i is independent of the data and will be replicated R_i times. Secondly, by basing the speed up limit calculation on the largest fraction of computation, perfect overlap between processes is assumed. Therefore, this speed up limit is only a very gross estimate of the potential performance of an implementation.

5.1 Description of iPSC/1 Implementation #1

The first implementation of BMDSIM was designed to isolate each BMDSIM function on a separate processor. The decomposition is a purely functional decomposition, and communications traffic was expected to be very heavy. Since approximately 50% of the computational load for BMDSIM occurs in one function, performance of this implementation was expected to be poor.

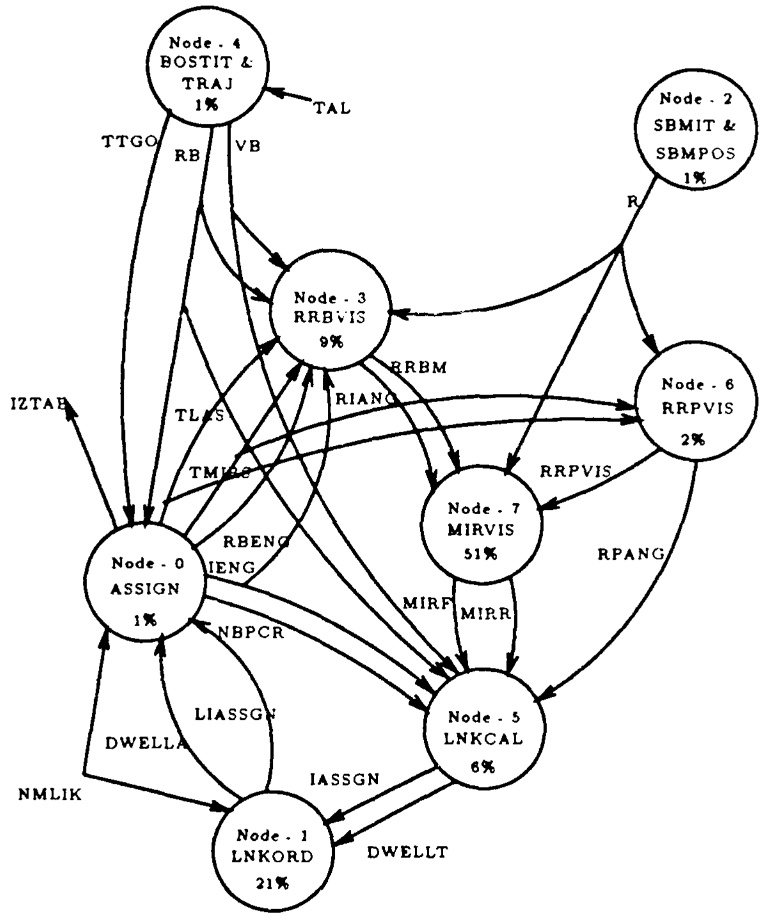

Figure 5.1. iPSC/1 Node Assignments and Communication for Implementation #1

5.1.1 Decomposition Process This decomposition was a completely functional decomposition of the original simulation. The main reasons for creating this decomposition were to study the performance of BMDSIM's sequential functions and the effect of completely distributing the control structure of the sequential simulation. Figure 5.1 represents the functional breakdown of this simulation, showing communication patterns, node mappings, and approximate percentage of computational load for the simulation processes, excluding the host process and its initialization messages. No attempt was made to limit the number or size of messages in the system. Since the DESE Research and Engineering functions were left intact, these functions were used to partition the simulation for parallelization. This decomposition maintains the deterministic behavior of the sequential simulation.

5.1.2 Parallelization Characteristics The major factors effecting the anticipated performance of this simulation are:

- Increased overhead to initialize the system, including time to load processes into nodes and to pass initialization data to node processes.
- Added overhead for interprocess communication between the nodes.
- Removed feedback loops associated with global data structures from within time steps, which caused continued computation after resources were no longer available.
- Majority (approximately 51%)of computations are performed in a non-replicated process.
- Added inefficiency by spreading loop structures on multiple processors.

The speed up limit for this implementation, based on Equations 5.1 and 5.2 is approximately 1.7.

5.2 Description of iPSC/1 Implementation #2

This implementation of BMDSIM was designed to reduce both the number and size of messages passed from implementation #1. The decomposition is a purely functional decomposition, with the majority of the computations being performed by a single node. Computational overlap in this implementation occurs only between the determination of feasible assignments, and the consumption of resources during that assignment.

Figure 5.2 shows the assignment of BMDSIM functions to processes and node processors. In addition, the approximate percentage of computation time from the sequential program profiling is included, though the percent of sequential program control derived from the "main" routine is ignored.

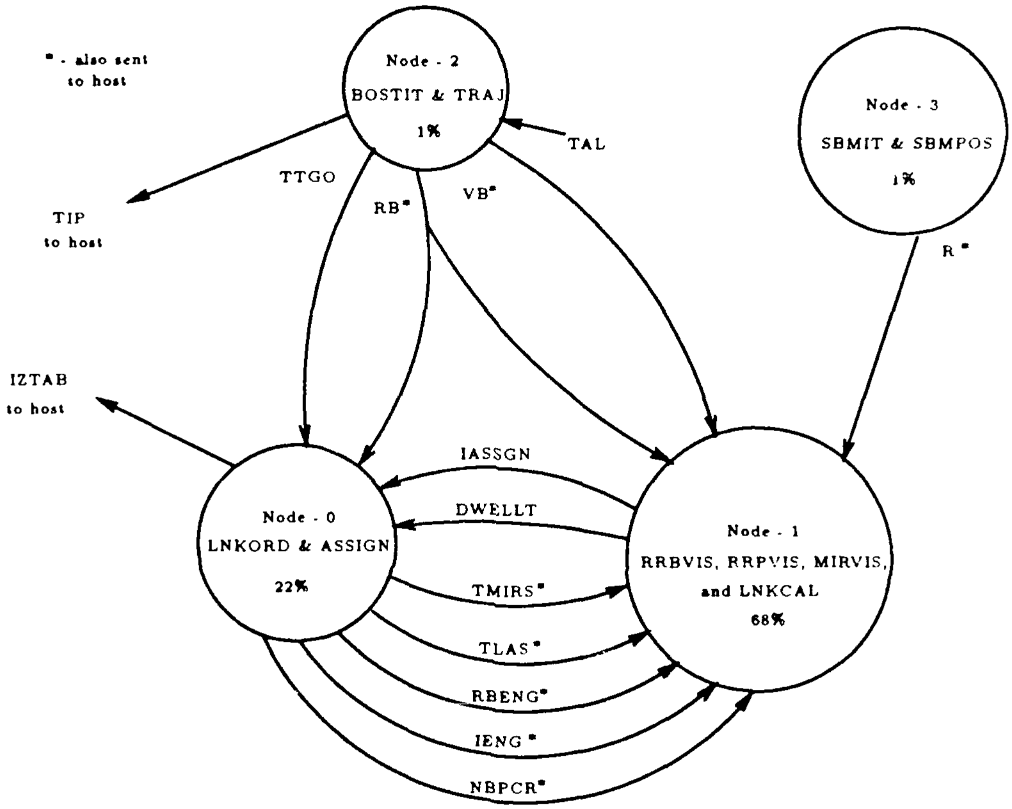

Figure 5.2. iPSC/1 Node Assignments and Communication for Implementation #2

5.2.1 Decomposition Process This decomposition was an attempt to reduce the number and size of messages in the system while maintaining a completely functional decomposition of the original simulation. Figure 5.2 represents the functional breakdown of this simulation, showing communication patterns and node mappings for the simulation processes, excluding the host process and it initialization messages. This decomposition maintains the deterministic behavior of the sequential simulation.

5.2.2 Parallelization Characteristics Two versions of this implementation were created. One version removed all feedback loops within each time step; the second supplied feedback as defensive elements were consumed (when there was an engagement in the time step). The removed feedback loops represented global data items used to control program computation in the sequential simulation. Neither version of this implementation was expected to execute in less time than the sequential version of the simulation. The primary reasons for this were:

- The increased overhead to initialize the system, including time to load processes into nodes and to pass initialization data to node processes.
- The added overhead for interprocess communication between the nodes.
- The removal of feedback associated with global data structures from loops within time steps, which caused continued computation after resources no longer available (first version only).
- The fact that a majority (approximately 68%)of computations are performed in a non-replicated process.

Additional messages were added to provide the time step synchronization required in a time driven synchronization since the number of clusters for which all calculations are performed is not constant. Using Equations 5.2 and 5.1, the limit of speed up for this implementation is under 1.3, $(\frac{1}{.77})$.

5.3 Description of iPSC/1 Implementation #3

Implementation #3 of BMDSIM was the first hybrid decomposition of the sequential program. The term "hybrid" indicates that it has characteristics of both functional and data decomposition. It represented an effort to combine major computational functions within a single process which could then be replicated. A decision was made at this point to allow non-determinism in the simulation. This decision was based on the methods previously discussed for maintaining determinism when processes are replicated, and the difficulties in implementing these methods on the iPSC/1 (limited memory, and the increase in coding complexity, see page 2-9). Assignments are made as feasible assignment information arrives at node 0 from nodes 5 through the total number of processors. No synchronization mechanism was created to guarantee node 0 processed this information in the same order as the sequential simulation.

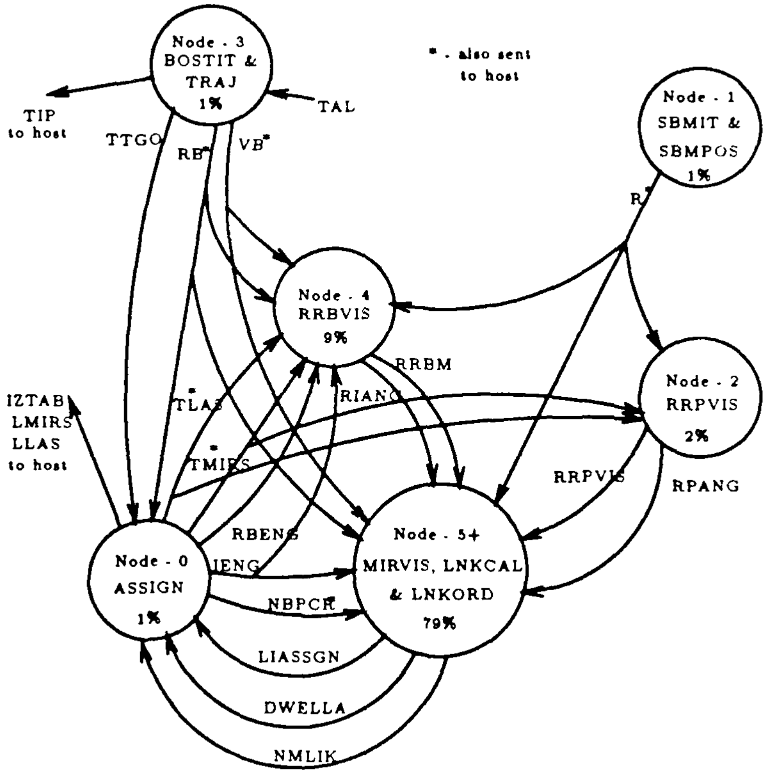

Figure 5.3. iPSC/1 Node Assignments and Communication for Implementation #3

5.3.1 Decomposition Process This hybrid decomposition required the creation of a mechanism for distributing computations to each of the replicated processes. Though the control structure for this is only marginally more complicated than the control structure required without a replicated process, the number of messages in the system is increased due to the increased number of processing nodes.

Since the iPSC/1 has a limit on the number of messages in the "sent but not yet received" state, intermediate messages in the simulation were combined where possible to decrease the amount of message traffic as much as possible. Figure 5.3 is a representation of this implementation showing approximate computational load and node mapping for simulation processes, excluding the host process and its initialization messages.

5.3.2 Parallelization Characteristics The primary differences between this simulation and the previous one are:

- The majority (approximately 79%)of computations are performed in a replicated process, increasing the amount of overlapping computations.
- Permitting non-deterministic behavior avoided additional synchronization, memory requirements, and control logic in system processes.
- Combined messages to reduce the number of messages in the system.

Since approximately 79% of the computational load for this implementation was contained in the replicated process, the limit for speed up determined by Equations 5.2 and 5.1 is 5.4 for both 32 and 16 processors, and 2.8 for 8 processors. The limit for 32 and 16 processors is the same because RRBVIS is the process which determines these limits.

5.4 Description of iPSC/1 Implementation #4

Implementation #4 of BMDSIM was also a hybrid decomposition of the sequential program. It represented an effort to reduce the number of messages in the system by replicating RRPVIS (laser-mirror range and visibility calculations). This function was selected due to the size of the data structures it generated. Each data structure generated by this function is 48,000 bytes in length, and iPSC/1 messages are limited to 16K bytes. In implementation #3, three messages were required

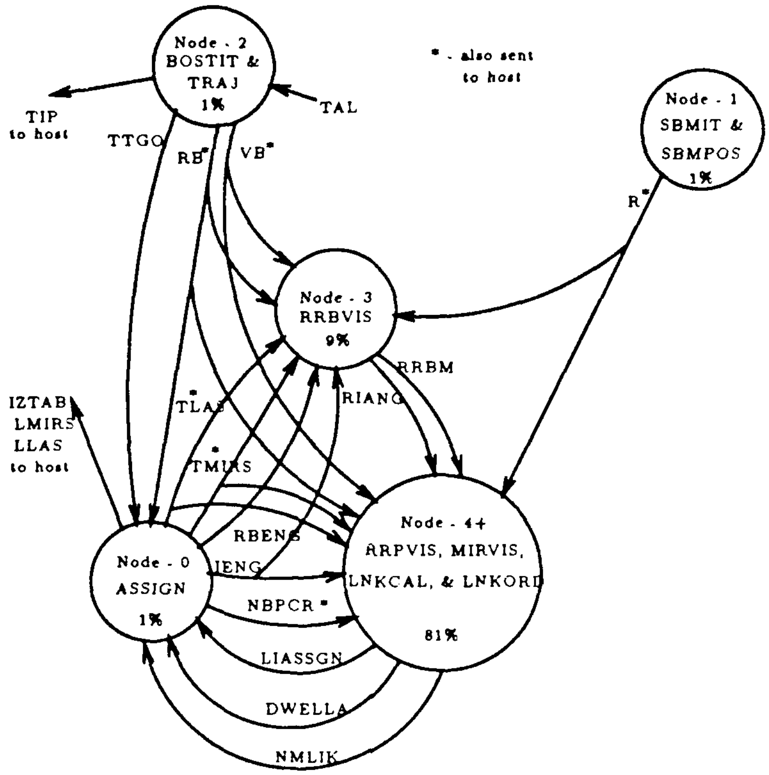

Figure 5.4. iPSC/1 Node Assignments and Communication for Implementation #4

to distribute each data structure to the other processing nodes which required the information.

5.4.1 Decomposition Process This decomposition was a direct modification of the previous implementation. No new messages were required, and few existing messages needed re-routing. A comparison of Figure 5.3 and Figure 5.4 reveals the similarities between these two implementations.

5.4.2 Parallelization Characteristics The primary difference between this simulation and the previous one is the reduction in the number of messages. The computational load for the replicated process increased by approximately 3% over the previous implementation, with the addition of RRPVIS. Approximately 82% of all computations are performed in the replicated process.

Since approximately 82% of the computational load for this implementation was contained in the replicated process, the limit for speed up determined by Equations 5.2 and 5.1 is 5.4 for both 32 and 16 processors, and 3.3 for 8 processors. As with implementation #3, the limit for 32 and 16 processors is the same because RRBVIS is the process which determines these limits.

5.5 Description of iPSC/1 Implementation #5

Implementation #5 of BMDSIM was another a hybrid decomposition of the sequential program. This decomposition extended the pattern of combining BMDSIM functions into a single replicated process. Message traffic was reduced by incorporating RRBVIS into the replicated process, and making a "supervisor" node process which allocated work to the replicated processes. This node process had no explicit equivalent in the sequential program, but incorporated some of the control structure within "main".

5.5.1 Decomposition Process This decomposition was a direct modification of the previous implementation. The major difference was the size and type of information included in the messages between the "supervisor" node and the replicated nodes. This message included only the number of boosters assigned to a replicated process, and the booster identifiers (indices into booster information data structures).

5.5.2 Parallelization Characteristics This implementation was expected to perform nearly as well as the previous implementation. With the addition of RRBVIS, the size and number of messages was reduced from implementation #4(only one message from the supervisor process to each replicated process for any given time step, rather than one message per booster cluster). However, the computational load for the replicated process increased by approximately 9% over the previous imple-

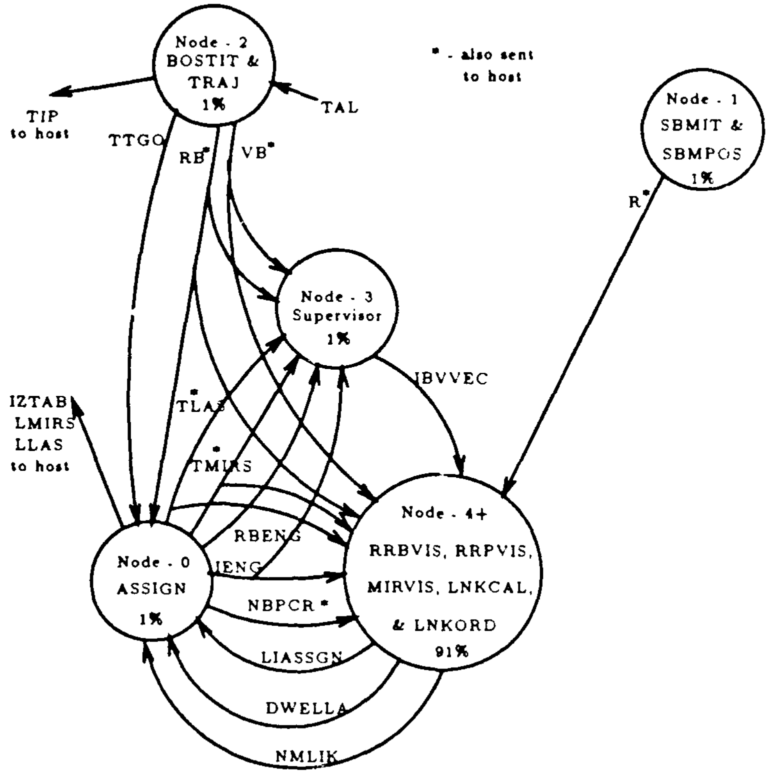

Figure 5.5. iPSC/1 Node Assignments and Communication for Implementation #5

mentation. Approximately 91% of all computations were performed in the replicated process.

The limits for speed up determined by Equations 5.2 and 5.1 are, 7.7 for 32 processors, 5.8 for 16 processors, and 3.1 for 8 processors. These are ideal limits which cannot be reached in practice.

5.6 *Description of iPSC/1 Implementation #6*

This hybrid decomposition of the sequential BMDSIM program combined additional functions into the replicated process of BMDSIM. Functions BOSTIT, TRAJ, SBMIT, and SBMPOS were added to the replicated process.

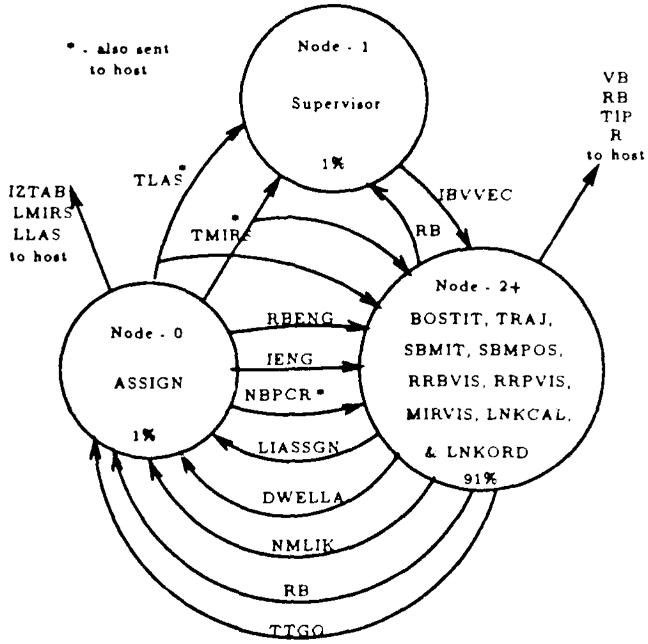

Figure 5.6. iPSC/1 Node Assignments and Communication for Implementation #6

5.6.1 Decomposition Process This decomposition was a direct modification of the previous implementation. All the booster and mirror position functions are distributed to the replicated processes. The other node processes received all booster information from the process on node 2. The messages were sent to the "supervisor" process, then ASSIGN node, and finally the host. This message traffic ordering was used overlap "supervisor" process computation with the communication time required to send the information to ASSIGN node and the host. The replicated process also sent mirror position data to the host. The "supervisor" and ASSIGN processes did not require mirror position information.

5.6.2 Parallelization Characteristics The computational load for the replicated process increased by approximately 1% over the previous implementation, with the addition of BOSTIT, TRAJ, SBMIT, and SBMPOS. Approximately 91% of all computations were performed in the replicated process. However, two additional processors are available for the replicated process, and message traffic at simulation clock synchronization was reduced.

The limits for speed up determined by Equations 5.2 and 5.1 are 7.9 for 32 processors, 6.2 for 16 processors, and 4.0 for 8 processors.

5.7 *Description of iPSC/1 Implementation #7*

This hybrid decomposition of the sequential BMDSIM program attempts to introduce more computation communication overlap in the simulation. Functions SBMIT and SBMPOS are placed in the process with ASSIGN, to overlap the communication of mirror position with the start of booster engagement calculations in the replicated processes.

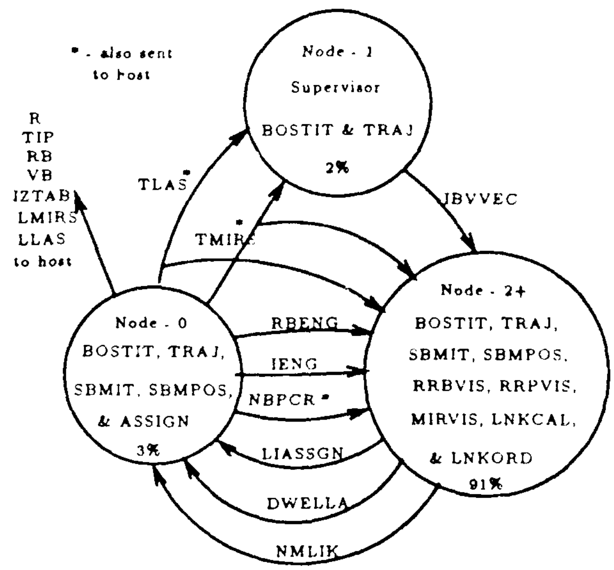

Figure 5.7. iPSC/1 Node Assignments and Communication for Implementation #7

5.7.1 *Decomposition Process*

This decomposition was a direct modification of the previous implementation. All the booster and mirror position functions are distributed to the processes requiring the position information. Mirror position functions were added to ASSIGN node, though mirror positions are not required on the node. This was done to make ASSIGN node the process which sends all position information to the host processor, because it does not interact with the replicated processes until they have completed processing of at least one booster cluster. This delay provided the time to send booster and mirror positions to the host.

5.7.2 Parallelization Characteristics All improvements for this implementation were the result of understanding when processes waited for communication, and what functions could be performed at those times. Each process required a synchronization mechanism to increment its local simulation clock. Since it was possible that a replicated process will have no boosters to perform calculations for, the synchronization message was required. Updated engagement information was sent at the end of each time step, indicating resource availability times for defensive elements. A process which completed calculations for all its boosters waited for this update information before proceeding with the next time step. In the previous implementation all processes waited for this information before proceeding. With this implementation, the supervisor computed its booster position data for the next time interval while the replicated processes determined new mirror positions before each waited for this update information.

The percent of processing performed by the replicated process is unchanged from implementation #6. The limits for speed up determined by Equations 5.2 and 5.1 are 7.9 for 32 processors, 6.2 for 16 processors, and 4.0 for 8 processors.

5.8 Description of iPSC/2 Version of Implementation #7

This implementation was a direct transfer of the iPSC/1 version. Any differences in performance between the two versions was a direct result of the differences between the method each system uses to pass message traffic (see Chapter 3). Speed up limits are independent of the architecture used for an implementation, and are the same as for the iPSC/1.

5.9 *Description of iPSC/1 Implementation #8*

This hybrid decomposition of the sequential BMDSIM program eliminates the supervisor process by statically allocating specific booster clusters to each replicated process.

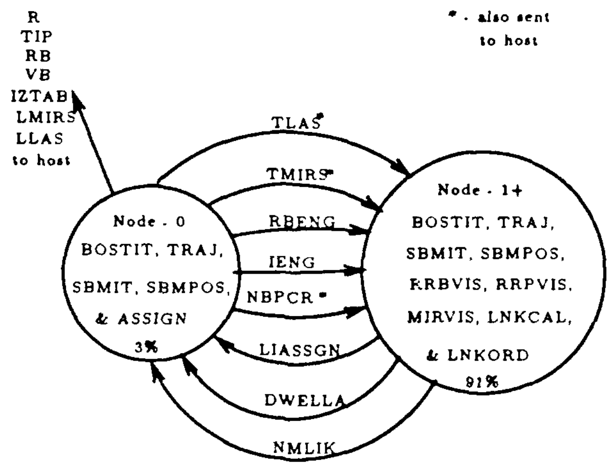

Figure 5.8. iPSC/1 Node Assignments and Communication for Implementation #8

5.9.1 Decomposition Process This decomposition was a direct modification of the previous implementation with the "supervisor" node eliminated. Each replicated process performed calculations for a subset of the boosters. No replicated process performed these calculations for adjacent booster indices. This represented an attempt to balance the computational load, based in part on the sample data set used. Adjacent indices tended to follow the same trajectories if they are of the same booster type and have nearly the same launch latitude and longitude. As a result, boosters with adjacent indices were often engaged in the same simulation time step, leaving a node with little or no computation for the next time step.

5.9.2 Parallelization Characteristics Speed up for this implementation was expected to be slightly less than for implementation #7. Though an additional processor was available for replicated processes, as the simulation progressed, load balance would decrease. In the worst case, some processes would have no computations to perform while others would never have any boosters engaged (destroyed) and would continue to perform all their calculations.

The limits for speed up determined by Equations 5.2 and 5.1 are 7.9 for 32 processors, 6.3 for 16 processors, and 4.4 for 8 processors.

5.10 Encore Implementation for the BMD Simulation

5.10.1 Implementation Description This implementation relied on the parallel constructs in the Encore Parallel FORTRAN language. Each data parallel "DO" loop in the sequential program was implemented as a parallel "DOALL" construct. Figure 5.9 provides a graphical depiction of the "parallel" sections of code, and the sequential sections, where shared memory is updated.

5.10.1.1 Decomposition Process The Encore Parallel FORTRAN compiler provided automated parallelization of code. However, the existence of variables which were not indexed on the loop variables prevented the automated parser from recognizing the data parallelism in the loop over booster clusters in the sequential code. Therefore, this section of code was manually modified to create a parallel section of code with appropriate local variables. The ASSIGN function became a critical section of code because the values of the defensive elements needed to be shared over all processors.

5.10.1.2 Parallelization Characteristics This implementation was equivalent to iPSC/1 implementation #7 in that feasible laser-mirror-booster assignments were determined and ordered in independent processes, while resources were committed in a "critical section" of code. This critical section of code is identified by using the Encore Parallel FORTRAN construct "Critical Section". Conceptually, iPSC implementation #7 and this Encore implementation are identical with the critical section equivalent to iPSC Node 0 process and the parallelized "DOALL" loops equivalent to all other iPSC processes. The general strategy of the simulation was to compute the feasible assignments for each booster in parallel and then to commit resources in the critical section of code.

The speed up limit figures for the Encore implementation will prove to be an even less accurate predictor of performance if Equations 5.1 and 5.2 are used. This is due to the isolation of parallel computations from sequential computations in these equations. Speed up limit figures for a shared memory processor can, instead, be estimated by

$$\begin{pmatrix} Shared \\ Memory \\ Speed\ up \\ Limit \end{pmatrix} = \frac{1}{\sum_{i=1}^{p} \frac{S_i}{R_i}} \tag{5.3}$$

Where S_i is the fraction of total computation performed by process i during sequential execution, R_i is the number of processors performing replicated process i, and p is the number of processes. For processes not in parallel sections of code, N_i is unity. Table 5.1 summarizes the results of Equation 5.3 for BMDSIM on the Encore.

The absolute speed up limit for the simulation, using Equation 5.3 is 8.991, and is determined by the maximum number of missiles in the simulation (NBOSTR = 100). Theoretically, this implies a speed up of 9 can never be achieved with this simulation, regardless of the number of processors applied to the problem.

5.1?.1.3 Exploitation of the Hardware The Encore operating system provides a run time environment conducive to parallel processing. When a program is compiled using a "parallelizing compiler" the resulting machine code checks an environment variable at program startup, and creates the number of identical processes determined by this environment variable (up to the number of processors in the system). This provides each image of the process, with access to the shared vari-

Table 5.1. Encore Parallel BMDSIM Speed up Limits

Processors	Speed up Limit
1	1.000
2	1.817
3	2.496
4	3.070
5	3.561
6	3.987
7	4.359
8	4.687
9	4.978
10	5.238
11	5.473
12	5.685
13	5.837
14	6.011
15	6.169
16	6.315

ables, and eliminates the overhead required to create processes dynamically during run time. As each parallel section of code is entered (a "DOALL" construct), these processes are activated to perform a portion of the processing.

The shared memory architecture allowed program state variables to be shared, and assigned either in non-parallel sections of code, or in "critical sections" of the parallel code ("critical section" refers to those sections of parallel code which modify shared variables). No messages needed to be passed, and the critical sections of code were controlled via the lock mechanism described in Chapter 3.

5.11 A Final Note on Parallel Implementations

The apparent bottleneck for all implementations of BMDSIM is the process containing the procedure "MIRVIS". Though the decision was made to keep the original sequential functions intact in the parallel versions, a more realistic approach for functional decomposition of this simulation would be to restructure all sequential functions to reduce the amount of computation performed in any single process. This restructuring implies changes to control and data structures used by each function to reduce the data dependencies. Any shared data items or nested loops would also be targets of any restructuring and rewriting of the original simulation.

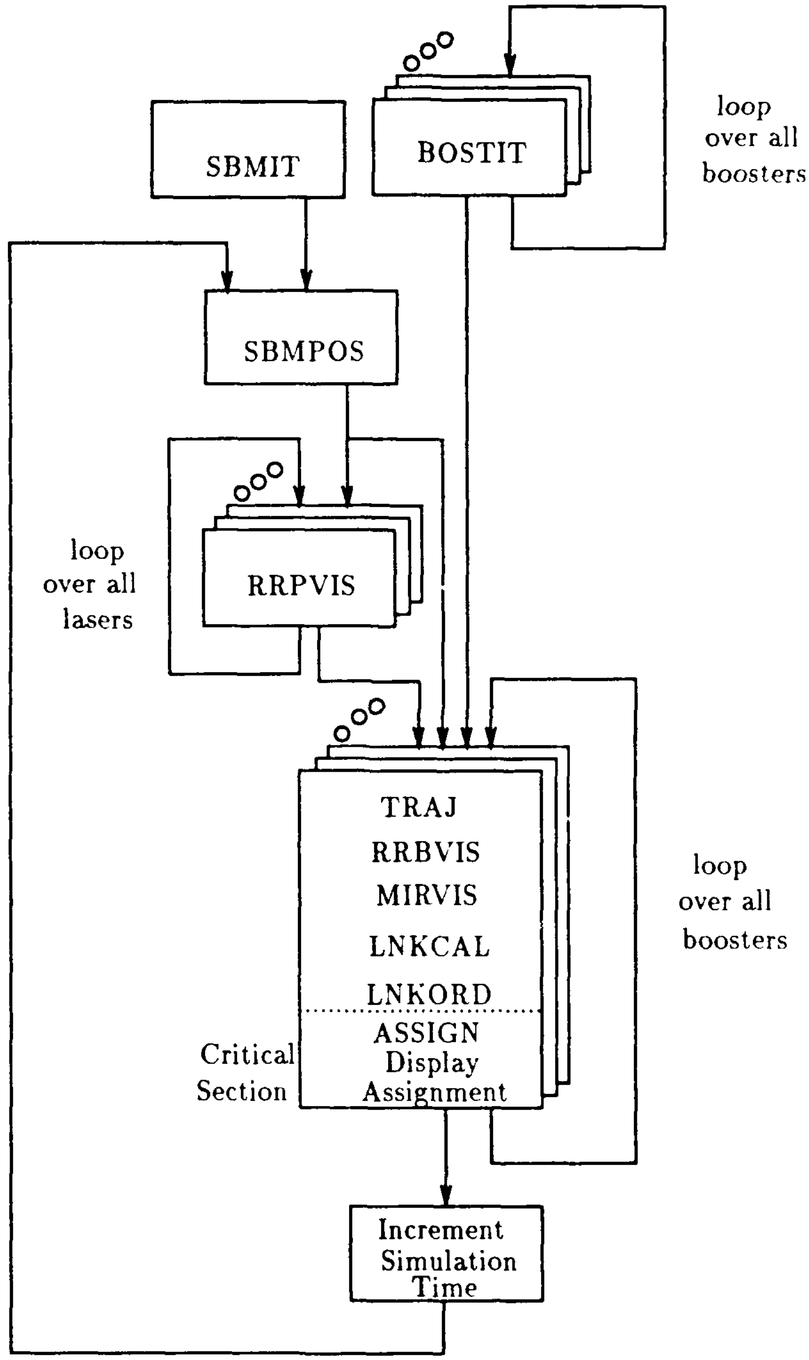

Figure 5.9. Encore - Implementation

VI. Empirical Results and Analysis

This chapter summarizes the results of each implementation of BMDSIM. All execution times used in computing speed up are average times for the execution of four or more trials of the simulation. Unless otherwise noted, execution times include all phases of processing, including loading processes into processor memory for the distributed memory systems and creation of processes in the shared memory system.

Speed up figures are derived from the execution of the sequential simulation executed on an equivalent processor. For the iPSC/1 and iPSC/2, the host processor was used to obtain sequential execution data. Since all Encore processors are identical, the sequential simulation was executed on a single Encore processor, after being compiled with the non-parallel FORTRAN compiler.

Overall execution timing information was collected using Unix system calls to determine the start and stop time of program execution. This provided a clock resolution of one second. The implementation overhead measurements were obtained from a one one-hundredth second resolution clock on the iPSC/1. No similar wall time function call was available on the iPSC/2 (host processor CPU time is the only available call); therefore, no overhead information was collected. Encore overhead measurements are obtained from a one second resolution wall time function call.

Though several iPSC/1 implementations were executed with a Sun 3 workstation operating as the "host" processor, these execution times are not included with the overall performance measurements in this chapter. The graphical interface provided by the Sun consistently required approximately one additional minute for initialization over the overhead figures for non-graphical trials. Aside from this additional overhead, there was no significant difference between trials using the iPSC/1 host processor and using the Sun 3 workstation as the "host".

An efficiency term is determined for each implementation, and is defined as

$$Efficiency = \frac{\text{Speed up}}{\text{Number of Processors}} \tag{6.1}$$

This efficiency value indicates performance relative to linear speed up.

6.1 iPSC Implementation Results

A complete set of results for all implementations is contained in Appendix C. Results for all iPSC implementations are summarized in Table 6.1. The speedup figures include all initialization overhead. The Eff_{limit} column in Table 6.2 represents the efficiency of the given trial with respect to the estimated limit from Equations 5.1 and 5.2. The limit value replaces the number of processors in Equation 6.1.

6.1.1 iPSC/1 Implementation #1 This implementation of BMDSIM has never run to completion. Based on partial executions of the implementation, a complete trial would require approximately 2.5 hours to execute. This implementation was not expected to perform well due to the additional overhead of message passing, and the use of FORTRAN functions from the original simulation as the logical processes implemented.

6.1.2 iPSC/1 Implementation #2 Program performance was as expected. The primary problem with this functional decomposition was the lack of overlapping processing. Based on the results of sequential program profiling, approximately 68% of the computational load is in a single process, and only 22% of all other processing can occur "in parallel" with this process due to data dependencies. The addition of feedback within simulation time intervals improved the overall performance of this implementation by reducing the number of wasted computations on the heavily loaded node.

Table 6.1. Summary of iPSC Implementation Results

Imple-mentation	Number of Pro-cessors	Overhead Time (seconds)	Speed up Limit	Speed up *vs.* Sequential	Efficiency	Eff_{limit}
#1	8	—	1.7	0.2	0.025	0.118
#2 no fdbk	4	56.88	1.3	0.532	0.133	0.409
#2a fdbk	4	58.28	1.3	0.988	0.247	0.760
#3	32	61.40	5.4	1.552	0.049	0.287
#3	16	60.96	5.4	1.916	0.120	0.355
#3	8	59.92	2.8	0.924	0.116	0.330
#4	32	56.92	5.4	2.644	0.083	0.490
#4	16	56.68	5.4	2.674	0.167	0.495
#4	8	55.80	3.3	1.834	0.229	0.556
#5	32	55.74	7.7	2.520	0.079	0.327
#5	16	55.50	5.8	1.906	0.119	0.329
#5	8	71.20	3.1	0.506	0.063	0.163
#6	32	45.88	7.9	3.430	0.107	0.434
#6	16	45.24	6.2	3.118	0.195	0.503
#6	8	44.36	4.0	1.834	0.229	0.459
iPSC/1 #7	32	52.16	7.9	4.529	0.142	0.573
iPSC/1 #7	16	50.62	6.2	3.428	0.214	0.553
iPSC/1 #7	8	49.72	4.0	1.762	0.220	0.441
iPSC/2 #7	8	—	4.0	1.929	0.241	0.482
#8	32	52.70	7.9	4.061	0.127	0.514
#8	16	50.86	6.3	3.057	0.191	0.485
#8	8	52.52	4.4	1.633	0.204	0.371

6.1.3 iPSC/1 Implementation #3 This was the first version of BMDSIM with a replicated process, and which produced non-deterministic results. One of the major performance considerations for the iPSC/1 is process-to-node mapping. This implementation was designed to dynamically determine the number of available processors, and to use that number. As a result, the number of multihop messages in the system increases as the number of processors increases (see description of message passing in the iPSC/1 on page 3-2). These multihop messages interrupted the processing on intermediate nodes, which reduced the amount of time spent working on the portion of data allocated to those nodes. This explains the degradation in performance between the 16 and 32 processor trials.

6.1.4 iPSC/1 Implementation #4 The additional 3% of computation added to this replicated process resulted in one additional processor being available for replicated processes. Speed up figures increased 70.4%, 39.6%, and 98.5% for 32, 16, and 8 processors respectively. However, the multihop messages still case degraded performance between the 16 and 32 processor trials.

6.1.5 iPSC/1 Implementation #5 This implementation was not expected to perform better than the previous one. The work load in the replicated process increased without a corresponding increase in the number of processors available to execute the replicated process. Approximately 91% of the computational load for this implementation was contained in the replicated process. The execution times measured while running on 8 processors varied from less than 30 minutes to more than 2 hours for the same program, data set, and host processor loads. The reason for this large variation in execution time is unknown.

6.1.6 iPSC/1 Implementation #6 Only three unique processes were identified in this implementation, allowing up to 29 processors to execute the replicated process. Approximately 92% of the computational load for this implementation was contained in the replicated process. The improvement in performance for this implementation over implementations #4 and #5 is the result of the trade-offs between computational load, communication, and number of processors for replicated processes. The increase in computation in the replicated process was offset by a reduction in communication and an increase in the number of processors available to execute the replicated process.

6.1.7 iPSC/1 Implementation #7 This implementation out performed implementation #6 because communication and computation timing was taken into account in the decomposition. This resulted in greater overlap between communication waits and computation in the node processes. As in implementation #6, approximately 91% of the computational load for this implementation was contained in the replicated process.

6.1.8 iPSC/2 Implementation #7 Speed up results are determined with respect to the sequential version of BMDSIM executed on the iPSC/2 host processor. The extra communications overhead on the iPSC/1 apparently accounted for approximately 9.5% percent of the execution time. This figure would be expected to increase as the number of processors in the simulation increases due to multihop message traffic. Since the only iPSC/2 system available has 8 nodes, it is not possible to check this conjecture.

6.1.9 iPSC/1 Implementation #8 This implementation was a static data decomposition of BMDSIM. As a result of the changing load balance of BMDSIM, this implementation will be load balanced only at the start of the simulation. As with implementation #7, approximately 91% of the computational load for this implementation was contained in the replicated process, which executed on all but one of the available node processors. The lack of load balance explains the lower speed up figures when compared to implementation #7, even though an additional processor is available to execute the replicated process.

6.2 Encore Implementation Results

Performance results for the Encore parallel implementation of BMDSIM are summarized in Table 6.2. Speed up results were determined with respect to both the sequential version of BMDSIM and to the parallel version executing on a single processor. For the hypercube implementations this comparison was not performed because changes to the number of processors used below the number of unique processes in each implementation meant a change to the source code.

Table 6.2. Encore Parallel BMDSIM Results

Number of Processors	Speed up *vs.* Sequential	Speed up Limit	Efficiency	$Efficiency_{limit}$
Sequential	1.000	—	—	—
1	0.919	1.000	0.919	0.919
2	1.449	1.817	0.725	0.797
3	2.195	2.496	0.732	0.879
4	2.364	3.070	0.732	0.879
5	2.770	3.561	0.554	0.778
6	3.219	3.987	0.537	0.807
7	3.661	4.359	0.523	0.340
8	3.292	4.687	0.412	0 702
9	3.336	4.978	0.371	0.670
10	3.826	5.238	0.383	0.730
11	3.634	5.473	0.330	0.664
12	4.056	5.685	0.338	0.713
13	3.900	5.837	0.300	0.668
14	4.105	6.011	0.293	0.683
15	4.024	6.169	0.268	0.652
16	4.428	6.315	0.277	0.701

The $Efficiency_{limit}$ column in Table 6.2 represents the efficiency of the given trial with respect to the estimated limit from Equation 5.3. The results in this column follow a generally decreasing pattern, which is as expected. As the number of processors increases, the likelihood of bus collisions and time spent waiting to enter the critical section of code, had a greater impact on the simulation performance. The speed up limit equations developed for shared memory architectures provide a more accurate indication of potential performance than those developed for distributed memory systems. A complete set of results is contained in Appendix C.

6.3 A Comparison of Architectures

6.3.1 Performance Figure 6.1 is a graph displaying the relative performance of the three architectures used in this effort, for a comparable implementation (iPSC implementation #7). The Encore Multimax provides more than 1.5 times the speed up than either of the iPSC implementations. When overhead figures are taken into account, as in Figure 6.2, the difference is less dramatic, but it still exists.

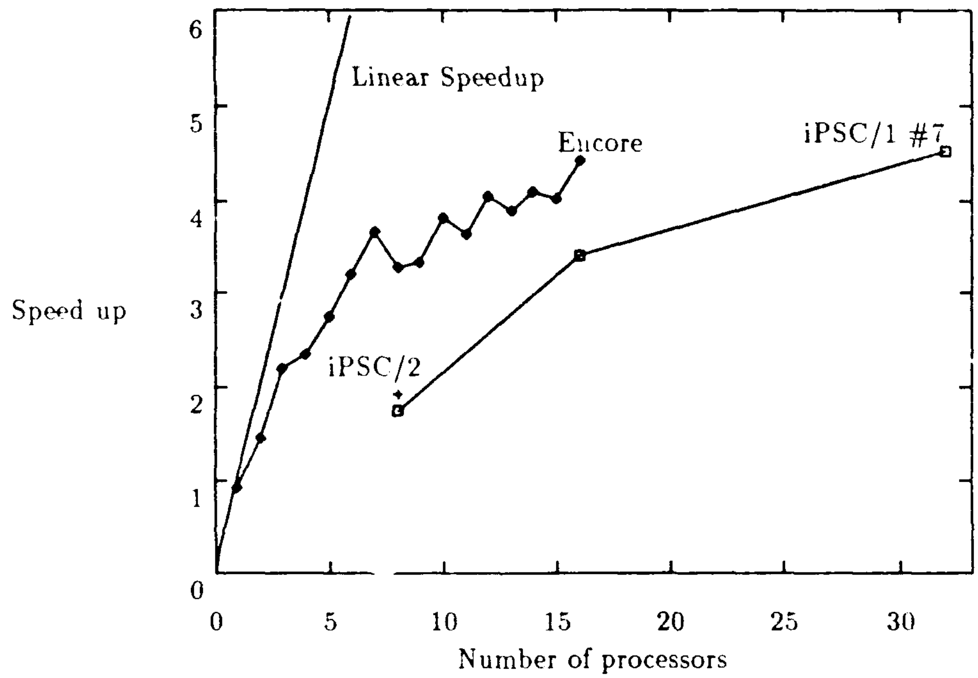

Figure 6.1. Speed up Graph for Comparable Implementations on Different Architectures

The difference in speed up between these architectures is determined by three major facto s. The times included in the speed up computations change the results obtained. Including overhead times in speed up calculations further distinguishes the performances obtained by the architectures studied here. This also provides a better comparison to the sequential version of the program by identifying all execution overhead incurred in parallelizing a program. The second factor affecting performance figures for these architectures is the difference between message passing

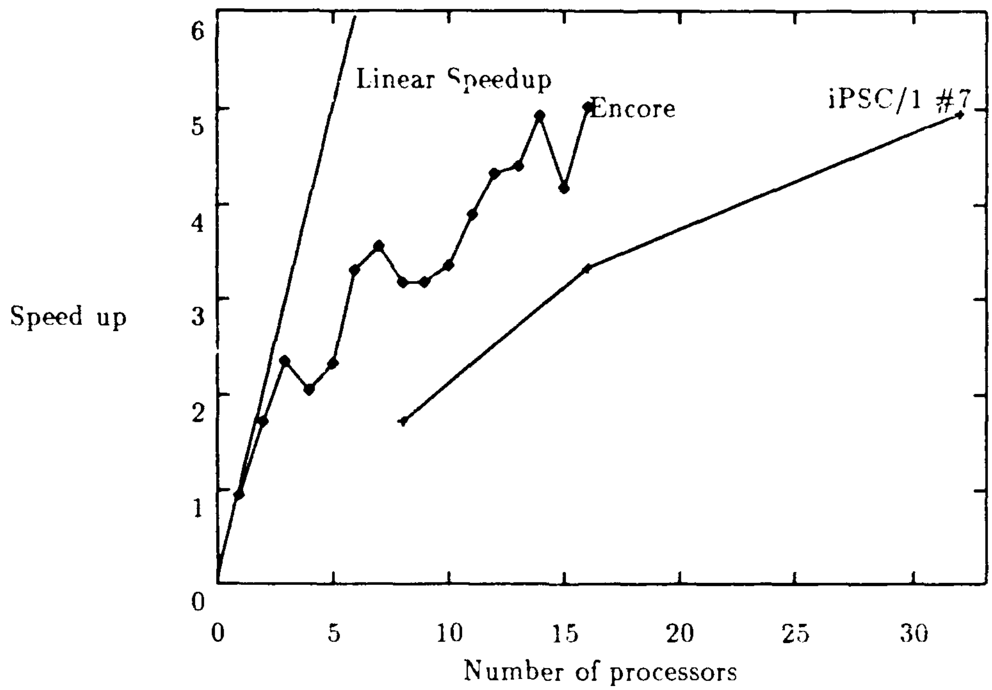

Figure 6.2. Speed up Graph for Comparable Implementations Excluding Initialization Overhead

overhead and bus contention overhead. Figure 6.3 represents the total overhead for each implementation in seconds.

The data used to plot Figure 6.3 are in absolute time (seconds), which amplifies the differences between the processor speeds. Figure 6.4 provides a comparison of these overhead times normalized with respect to their execution times. This graph indicates that overhead is primarily a function of the number of processors. While it would be tempting to conclude that the fraction of processing needed for overhead is independent of architecture, the accuracy of the time measurements became significant (approximately $\pm 2\%$ for 16 processors) for the Encore as execution time decreased.

The final factor is the time required to load programs into the appropriate

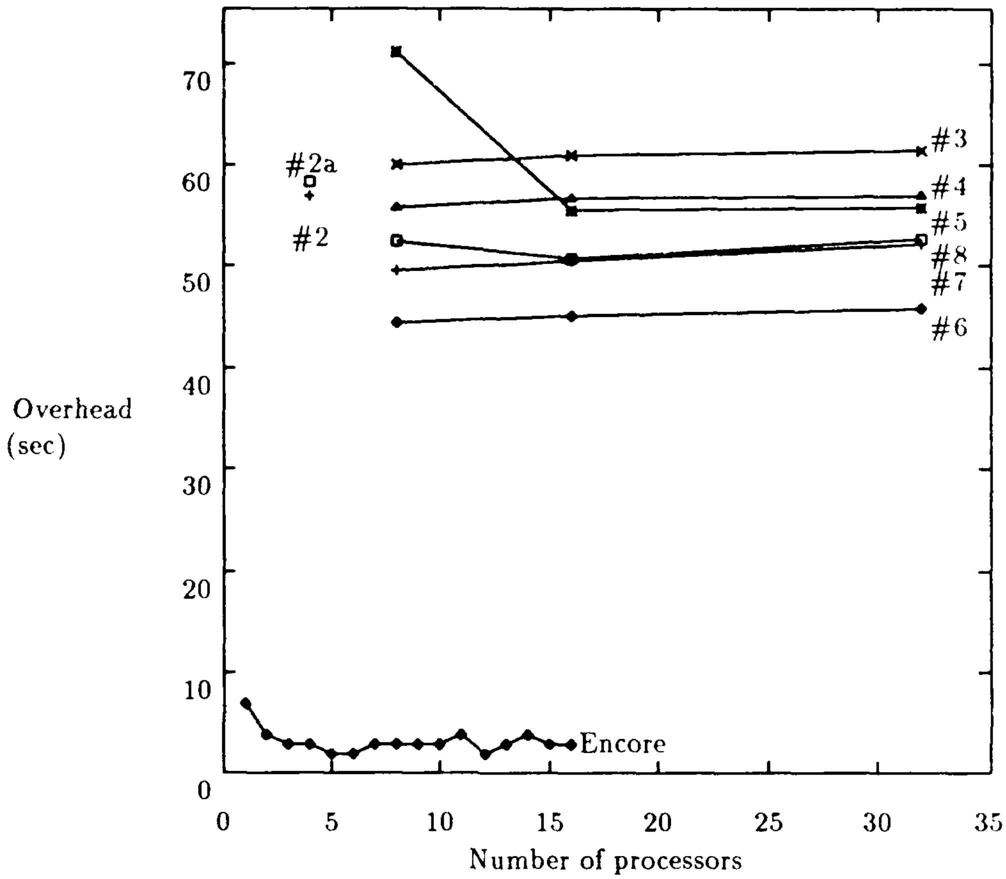

Figure 6.3. Overhead Times for Implementations

processor. Figure 6.5 is a graph the average of this time for all iPSC/1 implementations, and clearly shows this time is a function of the number of unique processes. The number of processors used does not affect this number in the iPSC systems because it is possible to load all node processors with a single process. No similar information is included for the Encore since it creates multiple processes at program initialization.

If there were a clear winner in this effort, it would be the shared memory architecture. Bus and memory contention had a negligible affect on performance

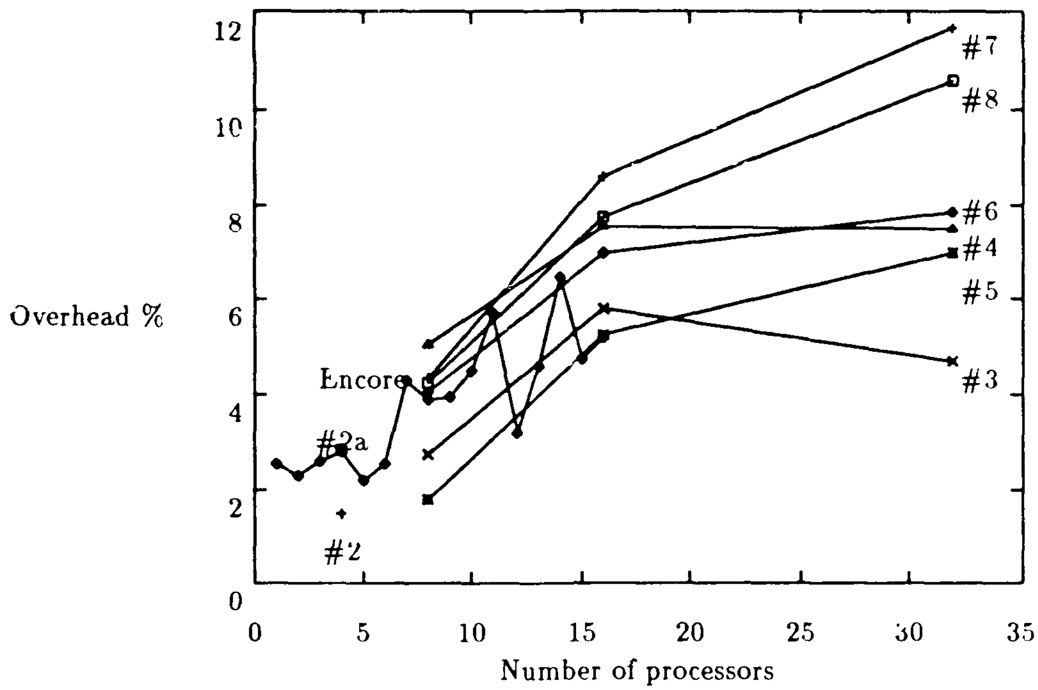

Figure 6.4. Normalized Overhead Times (% of Execution Time)

when compared to message passing overhead on both iPSC systems. This was due in part to the faster speeds of the "Nanobus" when compared to the Ethernet channel connections between nodes of the hypercube systems.

Shared memory architectures are limited in the number of processors which can be accommodated due to the increase in memory bandwidth requirements with the number of processors. The basic issue is what is "adequate" performance, and can it be achieved with the number of processors available in a shared memory system. Since distributed memory systems do not require the memory bandwidth of shared memory systems, greater numbers of processors can be applied to a given problem.

6.3.2 Programming Environment The shared memory system provided a much nicer programming environment. Many of the headaches of process control and

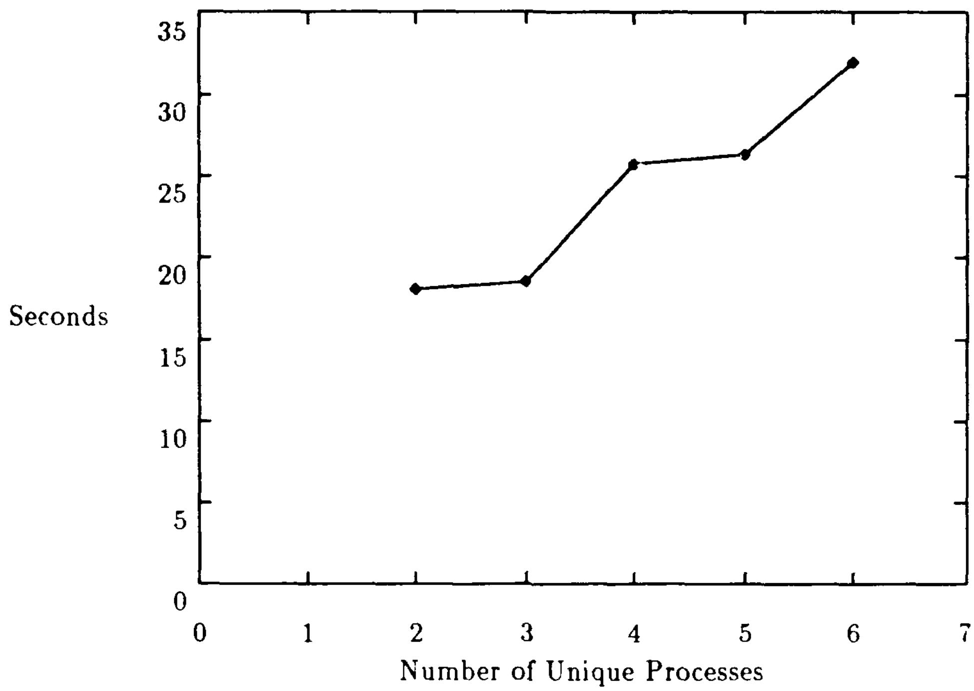

Figure 6.5. Average Time to Load Node Processors versus Number of Unique Processes

communication were handled by the operating system, since it must handle these problems for system processes. The parallel FORTRAN compiler removed process creation and communication an additional distance away by providing constructs to identify both parallel and critical sections of code. Program parallelization involved identification of the sections of code to be parallelized, the shared and local variables in those parallel sections of code, and any critical sections of code within the parallel sections. Program scalability to the number of processors was handled by setting a run time environment variable. There was no need to map processes to processors. Program execution was stable and predictable.

The iPSC systems, on the other hand, required creation processes, message traffic, and control structures to handle messages in the proper order. It was necessary to understand the limitations of the message handling system and the node processors in order to implement the simulation. Though a distributed memory system maps to functionally decomposed systems more easily (since it is based on the concept of separate communicating process), the iPSC systems were prone to inexplicable lock ups and unpredictable performance. Trials often would not execute to completion two times in a row, and system processing lights would indicate all nodes were waiting for communication, though BMDSIM will not deadlock unless message traffic is lost. In addition, though BMDSIM is non-deterministic in its computational load, the deviation in execution times between trials was much more than expected for implementation #5.

6.4 Guideline Development

Figure 6.6 shows the speed up graph, including initialization time, for all the iPSC/1 implementations. The relative performance of each implementation generally improves as the number of processors increases. This is very much as expected, and indicates data parallel programs possess greater potential for parallelism, than functionally decomposed programs. The primary reason for this is that a large data parallel simulation will generally provide a greater number of processes after data decomposition than after functional decomposition. A simulation with a large data set, where each element of the set is processed using the same functions, will be a good candidate for data decomposition.

The purely functional decompositions of this study show the affects of this property. Their speed up results are relatively poor, due to the dependencies and synchronizations needed for their implementation. However, as noted at the end of the previous chapter, the decision to keep the functions defined in the original simulation severely limited the performance of the functional decompositions. There is no reason to believe the relative performance obtained in this research is an inherent property of functional decompositions. However, since data decomposition can be

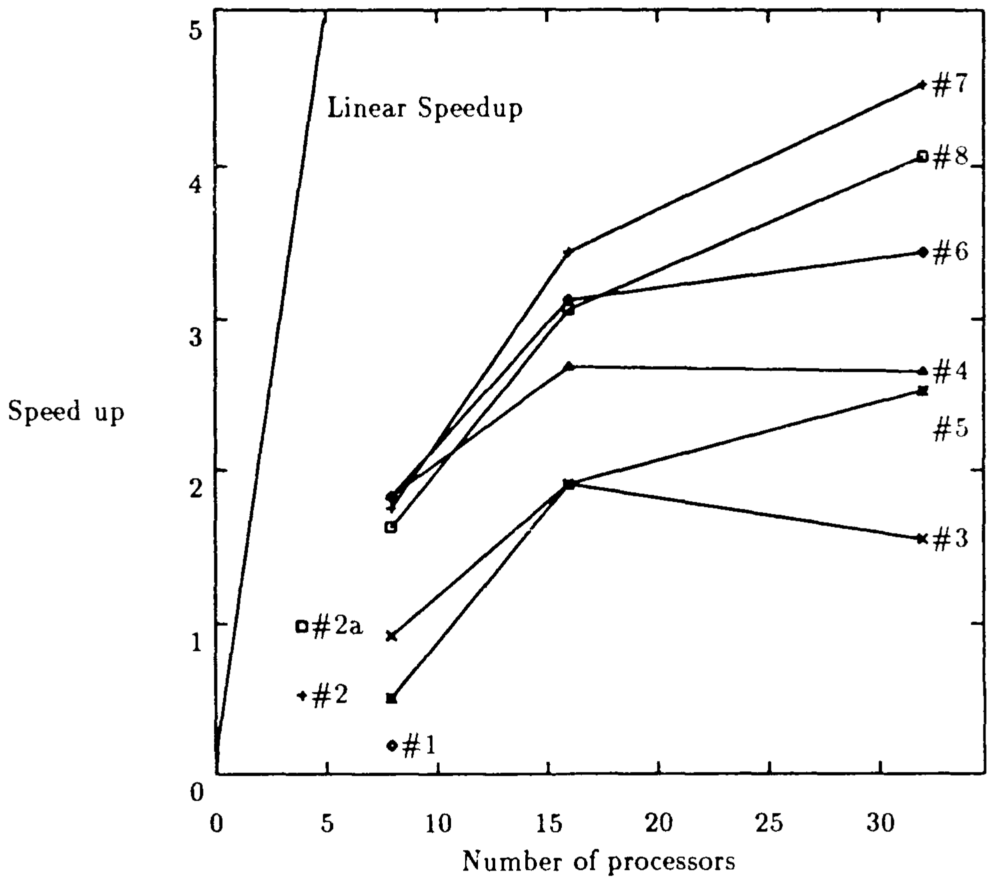

Figure 6.6. Speed up Graph for iPSC/1 Implementations

developed using existing functions, they generally require less effort to parallelize than functional decompositions.

Based on this information, parallelization guideline number one is:

Guideline One: Consider data decompositions before functional decompositions when the original simulation has a large decomposable data structure.

Another thing to note about Figure 6.6 is the relative performances of implementation #1 with and without feedback. This is case where, since feedback reduces the computational load, and the process receiving the feedback is computationally bound (approximately 68% of all processing), feedback is a good thing. Therefore, guideline number two is:

> *Guideline Two: Understand the effect of feedback on the computation before eliminating feedback loops.*

While not always an option, feedback loops which maintain current values of "global" data structures may be eliminated or reduced in frequency in order to reduce simulation message traffic.

The computational load in BMDSIM is not static, and depends on the available defensive elements and on the identification of boosters which have not been destroyed. Figure 6.7 shows the relative progress of the sequential simulation and implementation #7 on the iPSC/1. A constant slope line on this graph indicates a uniform computational load over simulated time.

The instantaneous speed up graph in Figure 6.8 represents the changes in computational load for BMDSIM and its affect on program parallelization efficiency.

The measurement of "instantaneous speed up" (proposed by Wieland) graphically depicts these changes in the computational load and the improvement provided by parallelization (41). Instantaneous speed up is defined as the ratio of derivatives

$$I_n(g) = \frac{dt_{seq}}{dt_n} = \frac{\frac{dt_{seq}}{dg}}{\frac{dt_n}{dg}} \tag{6.2}$$

where

$$g = \text{measured Global Virtual Time} = \text{Simulation Time}$$

$$t_{seq} = \text{real time for sequential simulation}$$

$$t_n = \text{real time for } n \text{ processors}$$

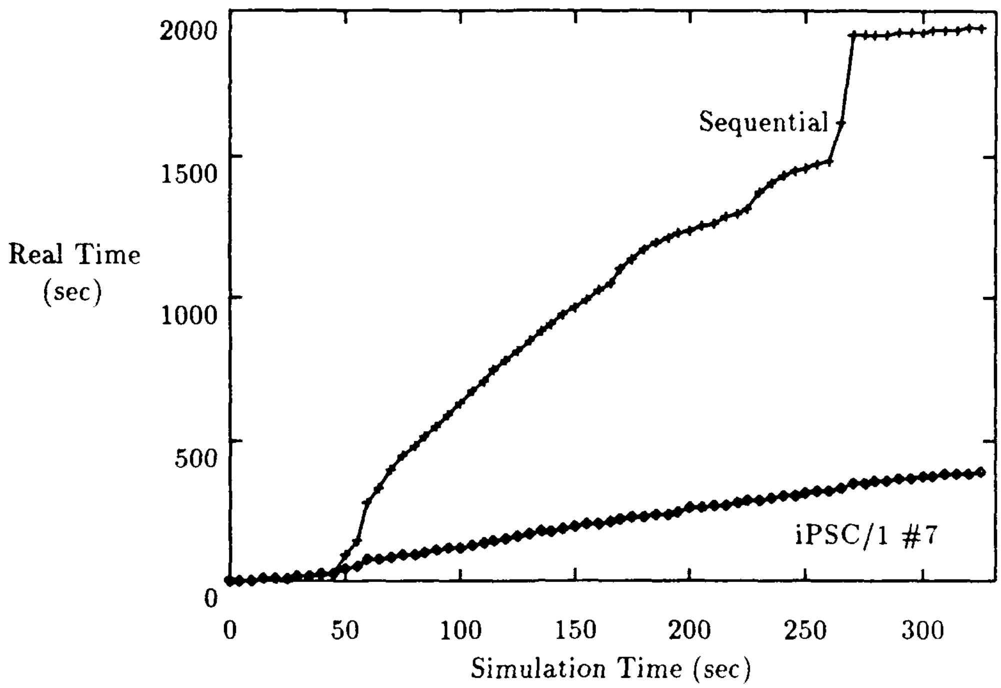

Figure 6.7. Progress of Sequential Simulation and Implementation #7 (32 nodes) on the iPSC/1

Though originally proposed as performance metric for simulations running under the "time warp" paradigm, instantaneous speed up reveals similar information about any parallel simulation where a global simulation time can be defined. The primary benefit of instantaneous speed up is as an indicator of portions of simulation time where speed up is limited (3).

The form of this graph follows expectations. The relatively low speedup regions at each end of the graph reflect those portions of simulated time where there are no booster clusters within the engagement window (that is they have not reach minimum engagement altitude, or they have burned out). The fluctuations in the central region of the graph correspond to the changes in computational load which can be attributed to relative availability of defensive elements at each increment of simulation time.

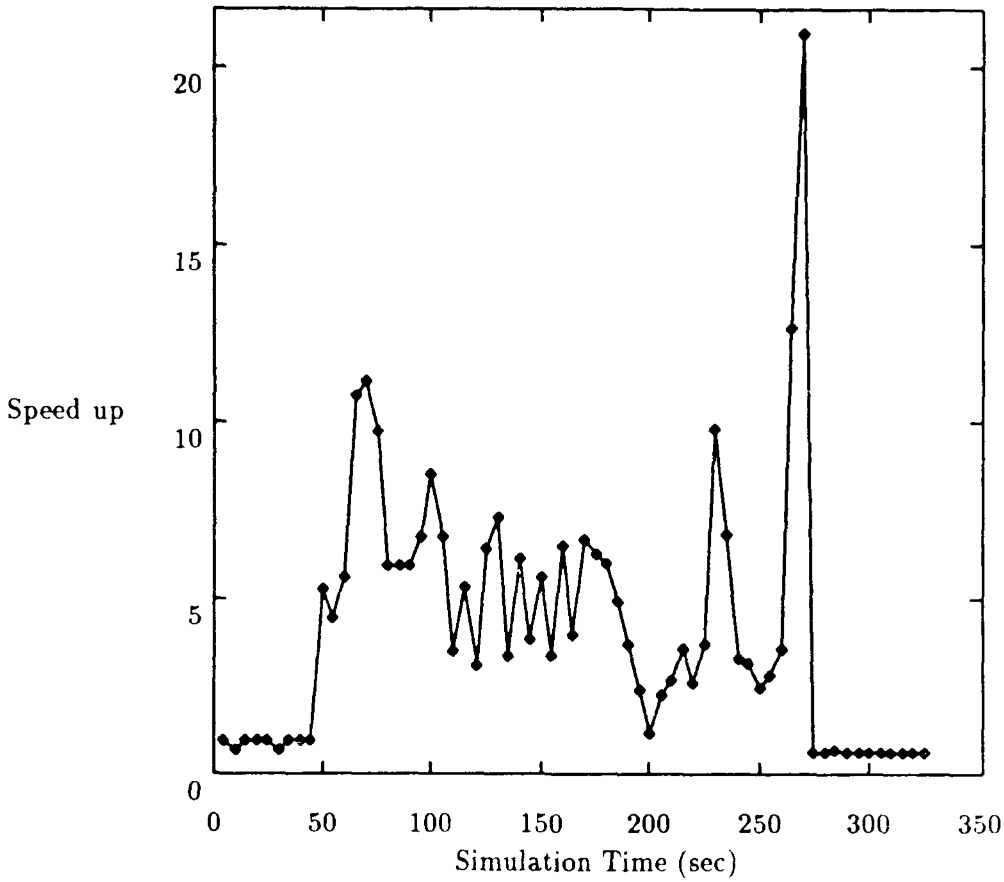

Figure 6.8. Instantaneous Speed up for one 32 node trial of Implementation #7 on the iPSC/1

The large spike at approximately 265 simulation seconds is a result of the engagement requirement that the engagement complete before cluster burnout. At that point in the simulation feasible links are determined for all remaining booster clusters, but few or no engagements meet this requirement. The variations in computational load for BMDSIM are not only time dependent they also depend on the data partitioning.

Figure 6.9 depicts the performance difference between static and dynamic data partitioning for BMDSIM. The result of a static data partition is that simulation load balance worsens as simulation time increases and processors have all their boosters destroyed. The efficiency of replicated processes is reduced by load imbalances, as displayed in Figure 6.10. The dynamic partition data is from implementation #7, with N-2 replicated processes, where N is the number of processors. The static partition data is from implementation #8, with N-1 replicated processes.

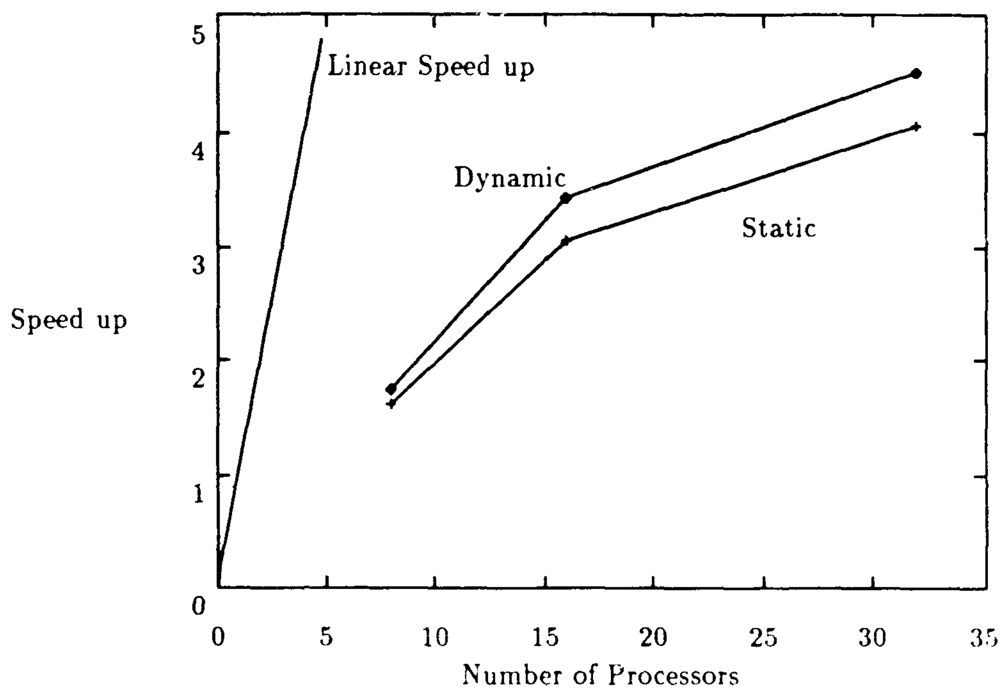

Figure 6.9. Dynamic versus Static Data Partitioning

A guideline appropriate for this observation is:

Guideline Three: When load balance varies with time over a partitioned data set, dynamic allocation of data items to processors will improve performance.

Note that this guideline is caveated by the ratio of replicated processes to data items in the partitioned data set. If data were partitioned with one data item per processor there would be no difference between static and dynamic partitioning.

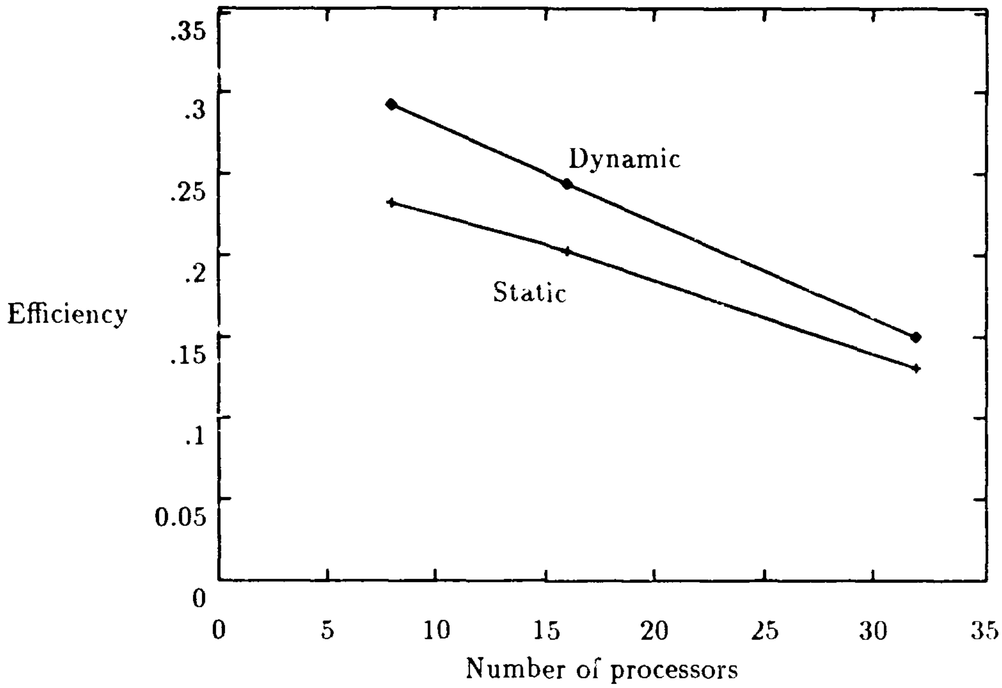

Figure 6.10. Replicated Process Efficiency in Static and Dynamic Data Partitioning

Other guidelines, which are not so readily apparent from the results, are derived from observations made while creating the implementations:

> *Guideline Four: Combine messages whenever possible; the number of messages tends to be more important than their size, and communication is more expensive than computation.*

Assuming the iPSC systems are representative of the class of distributed memory systems, communications overhead is extremely important. Combining message reduces system requirements between application processes, with regard to memory and, depending on the message passing mechanism, processing time. In addition, since most architectures are not fully connected, the effect of message traffic on intermediate processors must be minimized.

Guideline Five: Overlap communication waits with computation.

This guideline can be drawn directly from a comparison of the results for implementations #6 and #7 on the iPSC/1. If it is possible to predict when a process will be waiting for communication (for example, at a clock synchronization point in a time driven simulation), perform any computations possible for the next clock interval. This requires independence between the computation and the message in the pending communication.

Guideline Six: Minimize critical sections of code.

If a critical processing task or critical section of code requires a large fraction of processing time, this process will limit the speed up possible through parallelization. Therefore, in both distributed and shared memory systems, it is desirable to minimize critical sections of code in order to reduce the time for any process to occupy the critical section (for distributed memory systems critical code becomes a separate process or a synchronization message).

> *Guideline Seven: Determine whether the application requires determinism, and if not, what the performance trade-off is for determinism and non-determinism.*

This guideline is applicable to data decomposed systems which have a critical section of code. If determinism is required, mechanisms must be created to ensure that regardless of the order messages are received from replicated processes, the messages are processed in the "proper" order. These mechanisms will usually increase the space or processing requirements of parallelized program.

Guideline Eight: For existing simulations, understand how the decision to use existing code will limit parallelization options.

This guideline addresses the "speed up limits" computed for all implementations, and the partitioning options available when existing functions or subroutines are used. If one of the existing subroutines requires a large fraction of total processing time and cannot be replicated to provide data parallelism, that process is a limiting factor in the performance of any parallel implementation. In the case of BMDSIM, changes in the control structures and data structures combined with changes to MIRVIS, LNKCAL, LNKORD, and ASSIGN would have provided better speedup than changes to any single subroutine. On the other hand, the decision to redesign existing code to make it "more parallel" must be made with the understanding that the effort required in producing a parallel version of the program increases in proportion to the amount of code to be redesigned. In addition, the redesigned code will require more testing.

Guideline Nine: A simple model or equation for predicting performance can indicate the relative merit of two possible decompo... ns, but this is not an absolute indicator of performance.

The speed up limit model used in this effort provides only a rough estimate of potential performance. Such a model can be used to compare the relative merits of competing decompositions before the effort is expended in implementing them. Though it may be inaccurate in predicting the performance of any specific implementation, a model which provides results proportional to actual performance would be invaluable. No evidence exists to support any claims as to the speed up limit model's proportionality to actual performance. The accuracy of the speed up limit model is depicted in Figure 6.11 for the Encore implementation. Figure 6.12 shows the relative accuracy of the speed up limit model for the best iPSC/1 implementation.

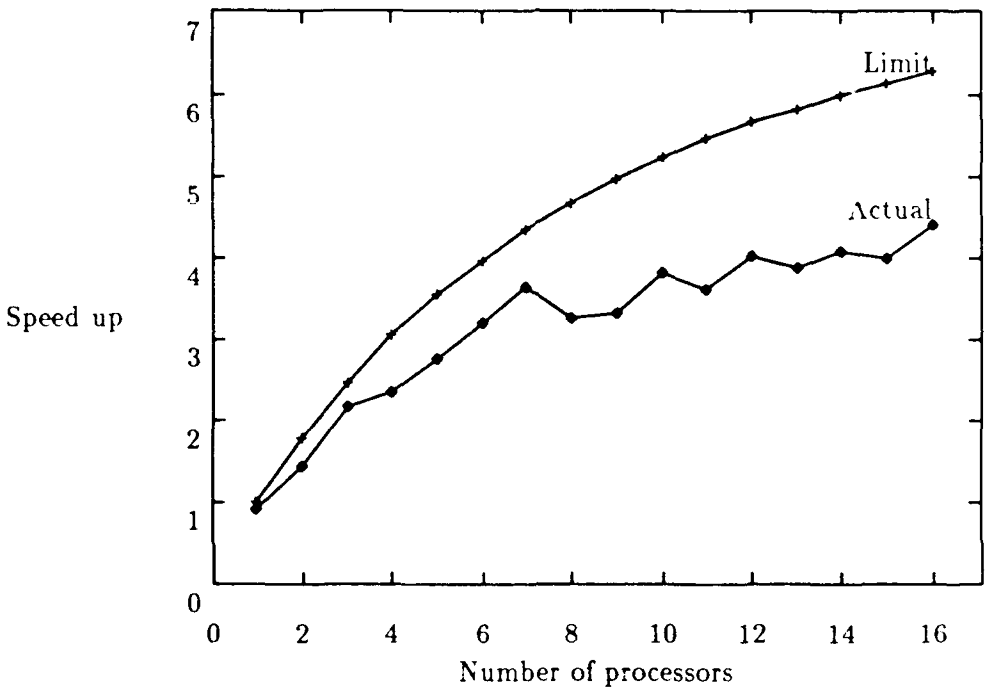

Figure 6.11. Actual Speed up versus "Speed up Limit" for the Encore

Finally, based on the architecture comparison of the previous section:

Guideline Ten: If program scalability to a large number of processors is not required, use a shared memory architecture.

Shared memory architectures will provide an easier environment in which to work. They will also have less overhead. However, current technology limits the number of processors in shared memory architectures to orders of magnitude less than distributed memory systems. Though not a major factor in this effort, some simulations

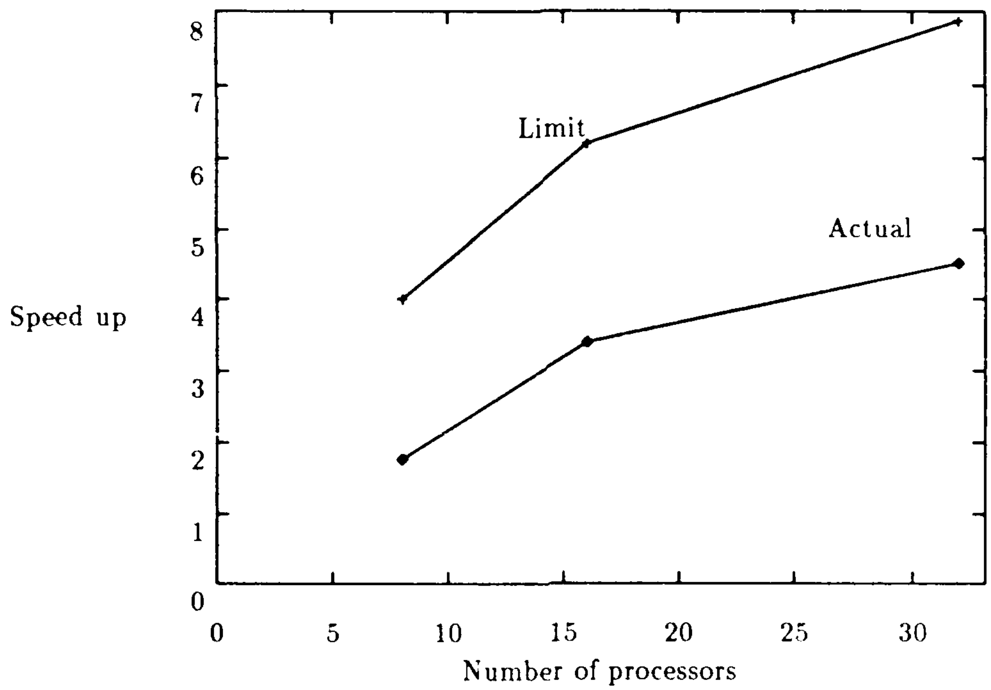

Figure 6.12. Actual Speed up versus "Speed up Limit" for iPSC/1 Implementation #7

are more logically suited to a message passing, distributed memory architecture with a large number of processors. The primary problem in these cases is finding a decomposition which balances the communication and computational requirements of the simulation. The large computational load in BMDSIM was sufficient to overwhelm the communication requirments during performance testing.

VII. Conclusions and Recommendations

7.1 Conclusions

Parallel processing offers the potential of increased program execution speed by distributing program computation across multiple processors. Though this aspect of parallel programming is widely accepted, there is no such agreement on the method or approach to take in distributing a program. When parallelizing an existing simulation program, additional issues must be addressed with respect to the amount of effort to be expended in the parallelization process.

This objective of this thesis effort was to develop a set of guidelines for parallelizing existing time driven simulations. The general approach was to use an existing time driven simulation to investigate the issues and options associated with program parallelization, and to use the empirical results from this investigation to develop the guidelines. The developed guidelines are a reflection of the simulation used to develop them and, until they have been validated by applying them to other simulations, they represent an analysis of the results obtained. The guidelines are contained in Appendix A.

The major accomplishments of this effort can be defined as follows:

- This effort provided the first documented "head-to-head" comparison of distributed memory architectures and shared memory architectures for a single application. The shared memory architecture used, though limited in the potential number of processors by memory bus bandwidth, provided a much better programming environment and less total system overhead. The results presented in Chapter 6 clearly demonstrate the advantages of shared memory architectures for comparable numbers of processors.
- Program decomposition is one of the major decisions to be made in program parallelization. The results of the various implementations for distributed

memory architectures demonstrated the advantages and disadvantages of data decomposed systems and functionally decomposed systems. The decomposition selected will determine the performance of the simulation, especially for a computationally intensive simulation. Though discrete event simulations were not investigated in this effort, this results should be equally applicable to discrete event simulations where each process acts on a similar data structure (the "colliding pucks" problem for example (4)).

- For distributed memory architectures, the method of dealing with "shared" memory structures in the sequential simulation can drastically affect the overall performance of the parallelized simulation. The "feedback" loops in the BMD simulation implemented shared data structures from the sequential simulation on the distributed memory architecture. The frequency of this feedback will influence not only program performance, but also program determinism, depending on the manner in which the shared structures are used (program control *vs* program computations)

- Program determinism becomes a major issue with parallel programs. A sequential program often derives much of its deterministic behavior from the control of the program instruction pointer. Sequential programmers often rely on this mechanism implicitly to provide deterministic behavior. When a program is parallelized this implicit control is absent, and it is up to the programmer to either accept the non-deterministic behavior of the parallelized program or to provide additional mechanisms to ensure deterministic execution. Determinism comes at the cost of memory space, additional computation, or both; but non-determinism introduces problems in program testing and accuracy.

- The entire parallelization process is a sequence of trade offs between performance and effort. Probably the largest single factor involved in this decision is the question of how much sequential code to use in the parallelized program. The decision to use existing code determines the control and data structures needed in the parallel program, limits the number of potential decompositions, and to a large extent will determine the maximum speed up any parallelized program will achieve.

The analysis involved in program parallelization can lead to the identification of deficiencies in the sequential program. If maximum program performance had been the goal of this thesis, much of the original source code would have been redesigned. Using original subroutines determined messages and control structures required in the parallel implementations, and reduced the options for parallelization and potential speed up.

7.2 Recommendations

There are many possible areas for continued study in the field of program parallelization. The following recommendations address some of the issues raised but not addressed in this thesis.

- Global memory structures in the sequential BMD simulation were maintained using feedback messages in the distributed memory system. Is there an efficient way to handle global memory structures in a distributed memory architecture? Sequential programs using global memory constructs for control or computation must have a means of maintaining these constructs when parallelized. What is the performance trade off? Can these structures be identified and localized to a particular process? What methods can be used to restructure a sequential program during parallelization to minimize the impact of such global structures?
- The speed up limit model presented in Chapter 5 provides only a very gross estimate of the relative performance of two parallel decompositions. A computational model which can be used to predict the relative performance of a

parallel program would be invaluable in reducing the effort involved in producing a "good" parallel decomposition of a program.

- The relative merits of static *vs* dynamic load balance were only partially mentioned in the results of Chapter 6. There is some conjecture that program load balance is determined by the process executing on a processor rather than the data that process is acting upon. Is dynamic data allocation a feasible solution in the cases where processes are replicated? If the process executing on a processor determines the load balance, what are the options for process migration, dynamic process creation and deletion, and other methods of reallocating processes to processors? What methods can be used to detect load imbalance?

- One of the major activities in parallelizing BMDSIM was analyzing the sequential simulation to determine what decompositions were feasible and which ones were likely to provide a performance improvement. The limitations of parallelizing compilers in recognizing program parallelism were briefly discussed in Chapter 2. Currently no automated tools exist to aid in identifying potential parallelism in sequential programs. A semantic analyzer capable of recognizing parallel constructs in general is probably not possible. However, a tool which could help this process would be an interactive parser for identifying variable usage and scope. For example, in BMDSIM all data structures in the FORTRAN "COMMON" blocks were manually identified for size and use in calls to subroutines and procedures. A tool which could parse the source code and determine the procedures requiring visibility to data structures would have helped to reduce the work load.

- The guidelines developed in this effort were based on results for a computationally bound time driven simulation. How applicable are the results to communication intensive or discrete event simulations (such as digital circuit simulation)?

- Program determinism is a major issue in parallel programming. Is there a way to quantify the cost of guaranteeing deterministic performance in a parallel program? What are the affects of determinism or its lack on simulation performance? Is there any rule for when deterministic behavior is required or desirable in program execution?

7.3 Summary

This research concentrated on the parallelization of time-driven simulation. While many questions remain in this field of study, the results presented here provide a baseline for the considerations and concerns to be taken into account when parallelizing a simulation. It is hoped that many of these issues and concerns can be generalized from the framework of the BMD simulation to the area of time-driven simulation and parallel simulation in general.

The largest factor to consider when parallelizing an existing simulation is to what extent existing code will be used. The decision to use existing code as a baseline for a parallel program will have an explicit impact on the parallel design when existing code is used in the parallel program. However, even if none of the original code is used, existing code used as a reference while designing a parallel simulation will influence many of the design decisions, even if only by example.

The guidelines developed in this research do not provide step-by-step instructions for parallelizing sequential simulations. The requirements and objectives of each simulation and the goal of the parallelization can vary so much between individual simulations that such specific instructions are not possible. Instead, these guidelines provide the programmer with issues to be considered and ideas on directions to take in parallelizing simulations.

Appendix A. *Guidelines for Simulation Parallelization*

A.1 General Concerns

Decisions must be made about the level of effort to be expended in parallelizing an existing simulation. There are many issues which will determine what level of effort is required while parallelizing the simulation. These issues are not independent and the decision made for one issue may affect the decisions made for several others. Of these issues, the following are among the most important:

- What is the purpose of the parallelization? If the goal of the parallelization is "maximum obtainable speed up", the decomposition decisions will probably differ from goals of "larger model size" or "moderate speed up and moderate effort".
- What is the expected performance improvement, and is it worth the level of effort?
- What architecture is best or is available for parallelization? The architecture used will influence the performance and level of effort of the parallelization. However, there is no choice involved when there is only one architecture available.
- Will the program be functionally decomposed, data decomposed, or will a combination of the two be more appropriate? Replicated independent data structures (such as array elements) must be present in the simulation if a data decomposition is desired. Functional decompositions are more likely to produce satisfactory results when independent operations are performed to obtain simulation results.
- Are there any critical sections of code? Shared memory structures introduce complexity into any decomposition. Ideally these structures would be limited

to one logical process, but this may not be possible in a data decomposed system. The method used to implement a critical section of code will affect the performance of a parallelized simulation.

- Is deterministic output required or desirable? Deterministic operation is guaranteed in sequential simulation. Parallel implementations of this same simulation may be non-deterministic depending on the algorithms and data structures used. If determinism is required the programmer must implement some mechanism to ensure deterministic operation.
- Does the program exhibit heavy computational loads or are computations simple? If computational loads are heavy in simulation processes communication overhead for distributed memory architectures will have less total impact on parallel simulation performance. Conversely, if computations are simple it may be advisable to combine logical processes onto single processors to reduce the impact of communication overhead.
- What general pattern of information or control flow does the program exhibit? A simulation with nested loops within the simulation time loop will complicate the parallelization process. If nested loops exist, it may be necessary to redesign that portion of the simulation or to combine the functions within the nested loops into larger logical processes.
- How much existing code will be used? This is one of the most important decisions to be made. This decision will impact both the overall performance of the parallel simulation and the level of effort required to parallelize the sequential program.
- Is the problem uniform in computation, or does computation vary with respect to simulation time? Wide variations in computational load between parallel processes will limit the speed up of the parallel simulation. If these variations can be predicted or detected, some load balancing technique may be employed

to redistributed the computational load, and improve performance. The decision to include load balancing in the simulation will increase the overhead of the parallel simulation, but may improve overall performance.

- Is the sequential simulation "good", and if not, will any effort be made to improve simulation algorithms during parallelization?

A.2 The Guidelines

The guidelines developed during this research are related to the issues raised in the previous section. They are not a set of step-by-step procedures for program parallelization. Instead, they are designed to spur thought on parallelization issues and to provide ideas on directions to take and methods to use in producing a "good" parallelization. General guidelines:

- *Guideline One: Consider data decompositions before functional decompositions when the original simulation has a large decomposable data structure.*
- *Guideline Two: Understand the effect of feedback on the computation before eliminating feedback loops*
- *Guideline Three: When load balance varies with time over a partitioned data set, dynamic allocation of data items to processors will improve performance.*
- *Guideline Four: Combine messages whenever possible, the number of messages tends to be more important than their size, and communication is more expensive than computation.*
- *Guideline Five: Overlap communication waits with computation.*
- *Guideline Six: Minimize critical sections of code.*
- *Guideline Seven: Determine whether the application requires determinism, and if not, what the performance trade-off is for determinism and non-determinism.*

- *Guideline Eight: For existing simulations, understand how the decision to use existing code will limit parallelization options.*
- *Guideline Nine: A simple model or equation for predicting performance can indicate the relative merit of two possible decompositions, but this is not an absolute indicator of performance.*
- *Guideline Ten: If program scalability to a large number of processors is not required, use a shared memory architecture.*

Basically the parallelization process is a trade off between the amount of work and time to be spent on a project and the speed up required or desired. While performance limits will exist, the decision as to what limits are or are not important will determine what metrics are used to call an implementation successful or a failure.

Appendix B. *BMD Simulation Data Structures*

Table B.1. BMD Simulation data descriptions and sizes

COMMON BLOCK STRUCTURES		
Name	Size (bytes)	Description
ITYPEA	400	vector (100), of booster cluster type used by BOSTIT and TRAJ
NBPC	400	vector (100), number of boosters for each cluster used to init NBPCR running tally, and in equation to determine fraction of leakers
TBL	800	vector (100), time before launch for each cluster (test data=0) used to determine vulnerability window for cluster
SEPAV	800	vector (100), average booster separation per cluster used in LNKCAL
XLATL	800	vector (100), launch latitude of cluster used in BOSTIT
XLONL	800	vector (100), launch longitude of cluster used in BOSTIT
XLATT	800	vector (100), target latitude of cluster used in BOSTIT
XLONT	800	vector (100), launch longitude of cluster used in BOSTIT
BGAM	800	vector (100), RV reentry flight path angle, used in BOSTIT (ORBEL)
VI	2400	array (3,100), initial velocity vector from cluster launch returned from BOSTIT and used in TRAJ
RL	2400	array (3,100), position vector of booster launch complex returned from BOSTIT and used in BMDSIM driver
RBO	2400	array (3,100), position vector of booster at burnout returned from BOSTIT and used in TRAJ
VBO	2400	array (3,100), velocity vector of booster at burnout returned from BOSTIT and used in TRAJ
TBO	2400	vector (100), time of booster cluster burnout returned from BOSTIT and used BMDSIM driver and TRAJ
RT	2400	array (3,100), position vector of target complex returned from BOSTIT and used in main
TFBOT	800	vector (100), time of flight from burnout to impact returned from BOSTIT and used in BMDSIM driver and TRAJ

Table B.2. BMD Simulation data descriptions and sizes (Continued)

COMMON BLOCK STRUCTURES - (Continued)		
Name	Size (bytes)	Description
AR	4800	array (3,3,100), transformation matrix to convert from coordinate system with X-axis along RL vector to one with X-axis in direction of Greenwich meridian at time of launch, returned from BOSTIT and used in TRAJ
AR	4800	array (3,3,100), transformation matrix to convert from coordinate system with X-axis along RL vector to one with X-axis in direction of Greenwich meridian at time of launch, returned from BOSTIT and used in TRAJ
RBERT	2400	array (3,100), a second positional vector for each cluster used by plotmis, with an inverted coordinate for image perspective
TIP	800	vector (100), time of impact used by graphics routine plotmis
RB	2400	array (3,100), position vector of booster cluster at time T returned from TRAJ and used in BMDSIM driver, RRBVIS, and LNKCAL (MIRGEO (VADD))
VB	2400	array (3,100), velocity vector of booster cluster at time T returned from TRAJ and used in RRBVIS, and LNKCAL (MIRGEO (VADD))
ALT	800	vector (100), altitude of booster cluster at time T
RANGE	800	vector (100), cluster range from launch
XLATP	200	vector (25), laser installation latitude used in POSVEC
XLONP	200	vector (25), laser installation longitude used in POSVEC
RP	600	array (3,25), laser position vector returned by POSVEC and used in RRPVIS (VMAG, VADD, and DOT) and LNKCAL (MIRGEO (VADD))
RSBM	8	radius of mirror orbit from surface of earth used in BMDSIM driver to set up value for SBMIT and SBMPOS
NSBMPO	4	number of space based mirrors per orbit used in SBMIT, SMBPOS, RRPVIS, RRBVIS, MIRVIS, and BMDSIM driver
NSBMO	4	number of space based mirror orbits used in SBMIT SMBPOS, RRPVIS, RRBVIS, MIRVIS, and BMDSIM driver

Table B.3. BMD Simulation data descriptions and sizes (Continued)

COMMON BLOCK STRUCTURES - (Continued)		
Name	Size (bytes)	Description
DELETA	8	true anomaly offset between the Kth mirrors in adjacent orbits, used in SBMIT
XINC	8	inclination of the mirror orbits to equatorial plane used in SMBPOS (SBMLOC)
RAO	80	vector (10), initial right ascension of the Jth mirror orbit returned by SBMIT and used in SBMPOS
ETAO	1600	array (10,20), initial true anomaly of the Kth mirror in the Jth orbit (J,K), returned by SBMIT and used in SMBPOS
RA	80	vector (10), right ascension of the Jth mirror orbit at time time T, returned from SBMPOS
ETA	1600	array (10,20),true anomaly of the Kth mirror in the Jth orbit (J,K), returned by SBMPOS
R	4800	array (3,10,20), position vector for each orbiting mirror at time T, returned from SBMPOS and used in RRPVIS, RRBVIS, MIRVIS, and LNKCAL
RRBM	1600	array (10,20), range between booster cluster and the Kth mirror in the Jth orbit (J,K), returned from RRBVIS and used in MIRVIS and LNKCAL
RBANG	1600	array (10,20), incident angle of laser on booster from Kth mirror in the Jth orbit (J,K), returned from RRBVIS and used in MIRVIS
RRPM	48000	array (10,20,30), range between Lth laser and the Kth mirror in the Jth orbit (J,K,L), returned from RRPVIS and used by MIRVIS
RPANG	48000	array (10,20,30), zenith angle between local vertical at the Lth laser and the Kth mirror in the Jth orbit (J,K,L), returned from RRPVIS and used by LNKCAL

Table B.4. BMD Simulation data descriptions and sizes (Continued)

COMMON BLOCK STRUCTURES - (Continued)		
Name	Size (bytes)	Description
MIRR	420	array (3,35), relay mirror data for the Ith geometrically feasible set of laser links, N = 1, orbit index J; N = 2, Mirror index K; N = 3, number battle mirror links for this relay mirror; (N,I), returned from MIRVIS and used by LNKCAL and LNKCK
MIRF	9800	array (2,35,35), battle mirror data for the Ith geometrically feasible set of laser links, K = 1, orbit index J; K = 2, Mirror index K; (K,I,M), returned from MIRVIS and used by LNKCAL (LNKCK)
IASGN	12000	array (4,25,30), an array containing the best mirror assignments for the Lth laser (I,J,L), returned by LNKCAL (DASET) and used by LNKORD
DWELLT	24000	array (4,25,30),an array containing the best dwell times including slewing and tracking, assignments for the Lth laser (I,J,L), returned by LNKCAL (DASET and MAXA) and used by LNKORD
LIASGN	1000	array (5,50), an array containing the best mirror assignments , returned by LNKORD and used by ASSIGN (SELECL)
DWELLA	1600	array (4,50),an array containing the best mirror dwell times, returned by LNKORD and used by ASSIGN (SELECL)
NBPCR	400	vector (100), number of remaining boosters per cluster, returned by ASSIGN and used by BMDSIM driver and LNKCAL
LMIRS	800	array (10,20), used by ASSIGN to count the number of boosters the Kth mirror in the Jth orbit is used against
LLAS	120	vector (30), used by ASSIGN to count the number of boosters each laser is used against

Table B.5. BMD Simulation data descriptions and sizes (Continued)

COMMON BLOCK STRUCTURES - (Continued)		
Name	Size (bytes)	Description
TMIRS	3200	array (10,20,2), returned by ASSIGN (SELECL), and used by BMDSIM driver, RRBVIS, RRPVIS, and LNKCAL to keep track of total time of mirror usage and next available time for assignment
TLAS	480	array (30,2), returned by ASSIGN (SELECL) and used by BMDSIM driver to keep track of total time of laser usage and next available time for assignment
IENG	800	array (10,20), the booster cluster last assigned the Kth mirror in the Jth orbit (J,K), returned by ASSIGN and used by RRBVIS and LNKCAL
RBENG	4800	array (3,10,20), position vector of booster cluster last assigned to the Kth mirror in the Jth orbit (V,J,K), returned by ASSIGN and used by RRBVIS (VADD, DOT, and VMAG)
SLANG	1600	array (10,20),slew angle between the present booster cluster and the previous booster cluster assigned for the Kth mirror in the Jth orbit (J,K), retuned by RRBVIS and used in LNKCAL

Table B.6. BMD Simulation data descriptions and sizes (Continued)

NOT DEFINED IN COMMON BLOCK		
Name	Size (bytes)	Description
SIGJR	24	vector (3), jitter for relay mirror I in micro-radians, used by LNKCAL (RELAY)
DALASM	2400	array (4,3,25), an array of intermediate results returned by LNKCAL (DASET) which are equivalent to DALAS not used
JDTIME	12	vector (3), used on the SUN workstation to display wall time
iztab	36	vector (9), used to display weapon assignment data on the SUN, returned by ASSIGN used by ZAP graphics routine
ORATE	8	orbital angular rate of mirrors in RAD/sec, returned by SBMIT and used in SBMPOS

Table B.7. BMD Simulation data descriptions and sizes (Continued)

CONSTANT DATA		
Name	Size (bytes)	Description
PI	8	3.1415926536D0
RAD	8	57.295779513D0 - used for coordinate conversions
RE	8	6378.16D0 - radius of the earth in Km
NMLS	4	50 - "maximum" number of lasers
NMIRL	4	5 - number of "best links to order in LNKORD and LNKCAL
IBMOP	4	1 - ASSIGN battle mgmt option, assign all boosters in cluster
IOPT	4	0 - RRBVIS and MIRVIS laser link filter option = 0, no filter = 1, do not consider links that have laser incident angles at the target less than "ANGMIN" (does not effect MIRVIS) = 2, same as iopt=1 plus selects the best "LMAX" links for further consideration based on minimizing the range squared divided by the sine of the incident angle (does not effect RRBVIS)

Appendix C. *Implementation Results*

This appendix contains tables summarizing the performance of the BMDSIM implementations.

Table C.1. iPSC/1 Implementation #1 Estimated Results

Number of Processors	Average Execution Time (sec)	Standard Deviation	Speed up *vs.* Sequential	Efficiency
Execution times including initialization				
Sequential	2013.25	30.752	1.000	—
8	approx. 9000	N/A	approx. 0.2	0.025

Table C.2. iPSC/1 Implementation #2 Results

Number of Processors	Average Execution Time (sec)	Standard Deviation	Speed up *vs.* Sequential	Efficiency
Execution times including initialization				
Sequential	2013.25	30.752	1.000	—
4 w/o feedback	3786.25	1.090	0.532	0.133
4 with feedback	2038.00	26.561	0.988	0.247
Typical Execution times without initialization				
Sequential	1962.48	No data	1.000	—
4 w/o feedback	3696.64	No data	0.531	0.133
4 with feedback	1923.58	No data	1.020	0.255

Table C.3. iPSC/1 Implementation #2 Overhead Time (seconds)

Program Version	Time to Load Nodes	Comm Time to Send Init Data	Wait time till Nodes Ready	Total time I/O & Init Overhead
Sequential	N/A	N/A	N/A	31.44
No feedback	26.02	7.64	25.64	56.88
With feedback	26.18	5.96	23.86	58.28

Table C.4. iPSC/1 Implementation #3 Results

Number of Processors	Average Execution Time (sec)	Standard Deviation	Speed up *vs.* Sequential	Efficiency
Execution times including initialization				
Sequential	2013.25	30.752	1.000	—
32	1297.50	18.621	1.552	0.049
16	1050.00	12.903	1.916	0.120
8	2179.25	16.300	0.924	0.116
Typical Execution times without initialization				
Sequential	1962.48	No data	1.000	—
32	1236.10	No data	1.588	0.050
16	989.04	No data	1.984	0.124
8	2119.33	No data	0.926	0.116

Table C.5. iPSC/1 Implementation #3 Overhead Time (seconds)

Number of Processors	Time to Load Nodes	Comm Time to Send Init Data	Wait time till Nodes Ready	Total time I/O & Init Overhead
Sequential	N/A	N/A	N/A	31.44
32	32.32	6.14	22.60	61.40
16	32.50	4.82	22.22	60.96
8	31.74	4.16	21.52	59.92

Table C.6. iPSC/1 Implementation #4 Results

Number of Processors	Average Execution Time (sec)	Standard Deviation	Speed up *vs.* Sequential	Efficiency
Execution times including initialization				
Sequential	2013.25	30.752	1.000	—
32	761.50	13.611	2.644	0.083
16	753.00	12.903	2.674	0.167
8	1097.75	2.681	1.834	0.229
Typical Execution times without initialization				
Sequential	1962.48	No data	1.000	—
32	677.90	No data	2.895	0.090
16	681.30	No data	2.880	0.180
8	1025.50	No data	1.914	0.239

Table C.7. iPSC/1 Implementation #4 Overhead Time (seconds)

Number of Processors	Time to Load Nodes	Comm Time to Send Init Data	Wait time till Nodes Ready	Total time I/O & Init Overhead
Sequential	N/A	N/A	N/A	31.44
32	26.68	4.20	22.42	56.92
16	26.68	4.92	23.58	56.68
8	25.88	4.22	24.24	55.80

Table C.8. iPSC/1 Implementation #5 Results

Number of Processors	Average Execution Time (sec)	Standard Deviation	Speed up *vs.* Sequential	Efficiency
Execution times including initialization				
Sequential	2013.25	30.752	1.000	—
32	799.00	59.068	2.520	0.079
16	1056.25	115.342	1.906	0.119
8	3975.75	2478.692	0.506	0.063
Typical Execution times without initialization				
Sequential	1962.48	No data	1.000	—
32	746.12	No data	2.630	0.082
16	1037.80	No data	1.891	0.118
8	2379.78	No data	0.825	0.103

Table C.9. iPSC/1 Implementation #5 Overhead Time (seconds)

Number of Processors	Time to Load Nodes	Comm Time to Send Init Data	Wait time till Nodes Ready	Total time I/O & Init Overhead
Sequential	N/A	N/A	N/A	31.44
32	25.58	6.24	22.46	55.74
16	25.54	4.84	23.66	55.50
8	40.48	4.22	24.26	71.20

Table C.10. iPSC/1 Implementation #6 Results

Number of Processors	Average Execution Time (sec)	Standard Deviation	Speed up *vs.* Sequential	Efficiency
Execution times including initialization				
Sequential	2013.25	30.752	1.000	—
32	587.00	7.280	3.430	0.107
16	645.75	29.482	3.118	0.195
8	1097.50	15.898	1.834	0.229
Typical Execution times without initialization				
Sequential	1962.48	No data	1.000	—
32	529.44	No data	3.707	0.116
16	595.46	No data	3.296	0.206
8	1104.68	No data	1.777	0.222

Table C.11. iPSC/1 Implementation #6 Overhead Time (seconds)

Number of Processors	Time to Load Nodes	Comm Time to Send Init Data	Wait time till Nodes Ready	Total time I/O & Init Overhead
Sequential	N/A	N/A	N/A	31.44
32	17.64	11.4	15.34	45.88
16	17.10	7.00	19.68	45.24
8	16.22	4.80	21.88	44.36

Table C.12. iPSC/1 Implementation #7 Results

Number of Processors	Average Execution Time (sec)	Standard Deviation	Speed up *vs.* Sequential	Efficiency
Execution times including initialization				
Sequential	2013.25	30.752	1.000	---
32	444.50	3.041	4.529	0.142
16	587.25	27.234	3.428	0.214
8	1142.75	31.901	1.762	0.220
Typical Execution times without initialization				
Sequential	1962.48	No data	1.000	—
32	396.55	6.909	4.949	0.155
16	587.25	27.234	3.342	0.209
8	1142.75	31.901	1.717	0.215

Table C.13. iPSC/1 Implementation #7 Overhead Time (seconds)

Number of Processors	Time to Load Nodes	Comm Time to Send Init Data	Wait time till Nodes Ready	Total time I/O & Init Overhead
Sequential	N/A	N/A	N/A	31.44
32	20.48	11.92	18.10	52.16
16	20.32	7.34	21.50	50.62
8	19.80	4.26	24.66	49.72

Table C.14. iPSC/2 Implementation #7 Results

Number of Processors	Average Execution Time (sec)	Standard Deviation	Speed up *vs.* Sequential	Efficiency
Execution times including initialization				
Sequential	279.25	0.829	1.000	---
8	144.75	4.815	1.929	0.241
Typical Execution times without initialization unavailable				

Table C.15. iPSC/1 Implementation #8 Results

Number of Processors	Average Execution Time (sec)	Standard Deviation	Speed up *vs.* Sequential	Efficiency
Execution times including initialization				
Sequential	2013.25	30.752	1.000	---
32	495.75	10.353	4.061	0.127
16	658.50	27.262	3.057	0.191
8	1232.50	12.894	1.633	0.204
Typical Execution times without initialization				
Sequential	1962.48	No data	1.000	---
32	435.92	No data	4.502	0.141
16	611.96	No data	3.207	0.200
8	1160.16	No data	1.692	0.212

Table C.16. iPSC/1 Implementation #8 Overhead Time (seconds)

Number of Processors	Time to Load Nodes	Comm Time to Send Init Data	Wait time till Nodes Ready	Total time I/O & Init Overhead
Sequential	N/A	N/A	N/A	31.44
32	16.98	16.65	16.74	52.70
16	19.36	7.88	22.14	50.86
8	18.26	7.36	24.42	52.52

Table C.17. Encore Parallel BMDSIM Results

Number of Processors	Average Execution Time (sec)	Standard Deviation	Speed up *vs.* 1 Proc Parallel	Speed up *vs.* Sequential
Sequential	253.50	1.118	—	1.000
1	275.75	0.433	1.000	0.919
2	175.00	21.90	1.576	1.449
3	115.50	9.014	2.387	2.195
4	107.25	6.220	2.571	2.364
5	91.50	6.874	3.014	2.770
6	78.75	4.603	3.502	3.219
7	69.25	4.023	3.982	3.661
8	77.00	2.345	3.581	3.292
9	76.00	2.550	3.628	3.336
10	66.25	2.947	4.162	3.826
11	69.75	5.262	3.953	3.634
12	62.50	2.291	4.412	4.056
13	65.00	2.915	4.242	3.900
14	61.75	8.526	4.466	4.105
15	63.00	2.739	4.377	4.024
16	57.25	2.861	4.817	4.428

Table C.18. Encore Parallel BMDSIM Efficiency

Processors	Efficiency	$Efficiency_{limit}$
1	0.919	0.919
2	0.725	0.797
3	0.732	0.879
4	0.591	0.770
5	0.554	0.778
6	0.537	0.807
7	0.523	0.840
8	0.412	0.702
9	0.371	0.670
10	0.383	0.730
11	0.330	0.664
12	0.338	0.713
13	0.300	0.668
14	0.293	0.683
15	0.268	0.652
16	0.277	0.701

Table C.19. Encore Parallel BMDSIM Overhead Times

Number of Processors	Typical Simulation Loop Time	Simulation Loop Speed up	Total Overhead Time
Sequential	246	—	7
1	263	0.935	7
2	143	1.720	4
3	104	2.365	3
4	120	2.050	3
5	106	2.321	2
6	74	3.324	2
7	69	3.565	3
8	77	3.195	3
9	77	3.195	3
10	73	3.370	3
11	63	3.905	4
12	57	4.316	2
13	56	4.393	3
14	50	4.920	4
15	59	4.169	3
16	49	5.020	3

Appendix D. *Program Pseudocode*

D.1 iPSC/1 Implementation #1

D.1.1 Host Program

Declare local variables and message types

Load node 0 program (assignment node - ASSIGN)

Load node 1 program (link ordering node - LNKORD)

Load node 2 program (mirror position node - SBMIT and SBMPOS)

Load node 3 program (booster-mirror visibility node - RRBVIS)

Load node 4 program (booster position node - BOSTIT and TRAJ)

Load node 5 program (feasible link node - LNKCAL)

Load node 6 program (laser-mirror visibility node - RRPVIS)

Load node 7 program (mirror-mirror visibility node - MIRVIS)

Open communication channel to nodes

Initialize graphics if Sun workstation host

Read number of clusters

    Swap byte order if Sun workstation host

    Send number of clusters to nodes 0, 3, 4, and 5

Loop over number of clusters

        Read cluster information (type, # boosters in cluster, time before launch, average separation between boosters, launch latitude, launch longitude, target latitude, target longitude, reentry flight path angle)

end loop

    Swap byte order if Sun workstation host

        Send cluster type, launch and target positions, and reentry angle
            to node 4

        Send average separation and number of boosters per cluster to nodes 3 and 5

Send time before launch and number of boosters per cluster to node 0

Read mirror information (Orbital distance from Earth's surface,

number of mirrors per orbit, number of mirror orbits,

true anomaly between mirrors in adjacent orbits, inclination

of mirror orbits to equatorial plane, mirror angular slew acceleration,

time to stablize mirrors and track booster)

Swap byte order if Sun workstation host

Send all mirror information to nodes 2, 3, and 5

Send number of orbits and mirrors per orbit to nodes 0, 1, 6, and 7

Read number of lasers

Swap byte order if Sun workstation host

Send number of lasers to nodes 0, 1, 5, 6, and 7

Loop over number of lasers

Read laser latitude and longitude

end loop

Read laser parameters (laser power output, beam quality, laser wavelength,

laser projector aperture, atmospheric divergence factor,

laser jitter, semi-major axis of relay mirror, semi-minor axis of

relay mirror, jitter for relay mirror 1, jitter for relay mirror2,

mean critical fluence required to kill the booster, standard

deviation of critical fluence, number of standard deviations

to be used in determining fluence and dwell time)

Swap byte order if Sun workstation host

Send laser parameters to nodes 5 and 6

Read Simulation increment, maximum simulation time, and flag

to turn on defenses

Swap byte order if Sun workstation host

Send increment, max time and flag to all nodes

START SIMULATION TIME LOOP

Receive pending message

Swap byte order if Sun workstation host

Case message type

MIMPACT – then received impact time and launch position vector for clusters, used in graphics routines.

MRPOSMSG – received mirror positions, plot if on Sun

MBUFMSG – received cluster positions, plot if on Sun

MSGZAP – received weapon assignment, plot if on Sun

LENGMSG1 – received number of boosters remaining in each cluster at end of simulation increment, increment time

END SIMULATION TIME LOOP

Output end of simulation information

D.1.2 Node 0 – ASSIGN

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive number of boosters per cluster

Receive number of mirrors per orbit, and number of mirror orbits

loop over number of mirror orbits

loop over number of mirrors per orbit

initialize mirror utilization arrays

end loop

end loop

Receive number of lasers

```
loop over number of lasers
    initial laser utilization array
end loop
Receive simulation time information
Receive time of cluster burnout
START SIMULATION LOOP
    Receive booster position for this time interval
    If not END OF TIME INTERVAL then
        Receive potential assignment information
        Call ASSIGN to determine which of the potential assignments
            can be made based on laser and mirror utilization data,
        if weapon assignment made then
            update laser and mirror utilization arrays and number of boosters
                remaining in cluster
            Send assignment to HOST
        end if
    else
        Receive message for end of simulation interval
        Send updated utilization information and number of remaining boosters
            per cluster to nodes 3, 5, 6, 7, and the host
        Increment simulation time
    end if
END SIMULATION LOOP
```

D.1.3 Node 1 - LNKORD

```
Declare local variables and message types
Open communication channels
```

Receive number of mirrors per orbit, and number of mirror orbits

Receive number of lasers

Receive simulation time information

START SIMULATION LOOP

  If not END OF TIME INTERVAL then

    Receive unsorted link information (3 messages)

    Call LNKORD to sort links by time to complete engagement

    Send "NMLIK" potential assignments to node 0- ASSIGN

  else

    Receive message for end of simulation interval

    Increment simulation time

  end if

END SIMULATION LOOP

D.1.4 Node 2 - SBMIT and SBMPOS

Declare local variables and message types

Open communication channels

Receive number of mirrors per orbit, and number of mirror orbits

Receive mirror altitude, true anomaly, and orbit inclination

Call SBMIT to initialize mirror orbits

Receive simulation time information

START SIMULATION LOOP

  Call SBMPOS to determine mirror positions for time interval

  If not first time interval then wait for END OF TIME INTERVAL message

  Send mirror positions to nodes 3, 5, 6, 7, and HOST

  Increment simulation time

END SIMULATION LOOP

D.1.5 Node 3 - RRBVIS

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive number of boosters per cluster and average separation

Receive number of mirrors per orbit, and number of mirror orbits

LOOP over number of mirror orbits

    LOOP over number of mirrors per orbit

        initialize mirror utilization arrays

    end loop

end loop

Receive simulation time information and defenses flag

Receive time of cluster burnout

START SIMULATION LOOP

    Receive mirror positions for this time interval

    Receive cluster positions for this time interval

    If defenses on then

        LOOP over booster clusters

            if boosters remain in cluster, before burn out, and

                above minimum altitude then

                Call RRBVIS

                Send RRBM, RIANG, and SLANG to nodes 5 and 7

                Receive reply from node 7 that ready for next cluster

        end loop

    end if

    Send END OF TIME INTERVAL message to nodes 0, 1, 2, and 4

Receive updated utilization information and number of remaining boosters

per cluster from node 0

Increment simulation time

END SIMULATION LOOP

D.1.6 Node 4 - BOSTIT and TRAJ

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive cluster types

Receive cluster launch and target positions, re-entry angle, and launch time

LOOP over number of clusters

initialize booster clusters determining position and

velocity vectors at launch, time of burnout, time

from burnout to impact, and coordinate conversion matrix

end loop

Receive simulation time information and defenses flag

START SIMULATION LOOP

LOOP over number of clusters

Determine cluster position and velocity vectors for this time interval

end loop

If not first time interval then wait for END OF TIME INTERVAL message

Send cluster position and velocity vectors to nodes 0, 3, 5, and the host

Increment simulation time

END SIMULATION LOOP

D.1.7 Node 5 - LNKCAL

```
Declare local variables and message types
Open communication channels
Receive number of clusters
Receive number of boosters per cluster and average separation
Receive number of mirrors per orbit, and number of mirror orbits
Receive mirror angular slew acceleration and time to stablize
    mirrors and track booster
Receive number of lasers
Receive laser parameters
Receive simulation time information and defenses flag
START SIMULATION LOOP
    Receive mirror positions for this time interval
    Receive cluster positions for this time interval
    Receive laser-mirror angle messages from node 6 (3 messages)
    if updated utilization and number of remaining boosters message has
            not been received
        Receive booster mirror ranges, beam incident angle, and
            mirror slew angle from node 3
        LOOP while receiving laser-mirror-cluster data for
                current cluster from node 7
            Receive laser-mirror-cluster data
            Call LNKCAL
        end loop
        if any lasers are available
            Send index into cluster data structures, and unsorted
                link information to node 1
        end if
    end if
```

Receive updated utilization information and number of remaining boosters
per cluster from node 0
Increment simulation time
END SIMULATION LOOP

D.1.8 Node 6 – RRPVIS

Declare local variables and message types
Open communication channels
Receive number of mirrors per orbit, and number of mirror orbits
LOOP over number of mirror orbits
LOOP over number of mirrors per orbit
initialize mirror utilization arrays
end loop
end loop
Receive number of lasers
Receive laser parameters
LOOP over number of lasers
initialize laser utilization arrays and laser position vectors
end loop
Receive simulation time information and defenses flag
START SIMULATION LOOP
Receive mirror positions for this time interval
if defenses are on then
LOOP over number of lasers
Call RRPVIS
end loop
Send laser-mirror ranges to node 7 (3 messages)

```
        Send laser-mirror angles to node 5 (3 messages)
    end if
    Receive updated mirror utilization times from node 0
    Increment simulation time
END SIMULATION LOOP
```

D.1.9 Node 7 - MIRVIS

```
Declare local variables and message types
Open communication channels
Receive number of mirrors per orbit, and number of mirror orbits
Receive number of lasers
LOOP over number of lasers
    initialize laser utilization times
end loop
Receive simulation time information and defenses flag
START SIMULATION LOOP
    Receive mirror positions for this time interval
    Receive laser-mirror range messages from node 6 (3 messages)
    if END OF TIME INTERVAL message has not been sent
        Update laser utilization times if update message received from node 0
        Receive booster mirror ranges, beam incident angle, and
            mirror slew angle from node 3
        LOOP over number of lasers
            if laser is available in this time interval
                Call MIRVIS
                Send laser number and mirror data to node 5
        end loop
```

Send total number of lasers available to node 5

Send message to node 3 to let it know ready for next cluster

end if

Receive END OF TIME INTERVAL message from node 3

Receive updated laser utilization times from node 0

Increment simulation time

END SIMULATION LOOP

D.2 iPSC/1 Implementation #2

D.2.1 Host Program

Declare local variables and message types

Load node 0 program (assignment node - LNKORD and ASSIGN)

Load node 1 program (link calculation node - RRBVIS, RRPVIS, MIRVIS, and LNKCAL)

Load node 2 program (booster position node - BOSTIT and TRAJ)

Load node 3 program (mirror position node - SBMIT and SBMPOS)

Open communication channel to nodes

Initialize graphics if Sun workstation host

Read number of clusters

Swap byte order if Sun workstation host

Send number of clusters to nodes 0, 1, and 2

Loop over number of clusters

Read cluster information (see implementation #1)

end loop

Swap byte order if Sun workstation host

Send cluster type, launch and target positions, and reentry angle to node 2

Send average separation and number of boosters per cluster to node 1

Send time before launch and number of boosters per cluster to node 0

Read mirror information (see implementation #1)

Swap byte order if Sun workstation host

Send all mirror information to nodes 1 and 3

Send number of orbits and mirrors per orbit to node 0

Read number of lasers

Swap byte order if Sun workstation host

Send number of lasers to nodes 0 and 3

Loop over number of lasers

Read laser latitude and longitude

end loop

Read laser parameters (see implementation #1)

Swap byte order if Sun workstation host

Send laser parameters to node 1

Read Simulation increment, maximum simulation time, and flag to turn on defenses

Swap byte order if Sun workstation host

Send increment, max time and flag to all nodes

START SIMULATION TIME LOOP

Receive pending message

Swap byte order if Sun workstation host

Case message type

MIMPACT - then received impact time and launch position vector for clusters, used in graphics routines.

MRPOSMSG - received mirror positions, plot if on Sun

MBUFMSG - received cluster positions, plot if on Sun

MSGZAP - received weapon assignment, plot if on Sun

LENGMSG1 - received number of boosters remaining in each cluster at end of simulation increment, increment time

END SIMULATION TIME LOOP

Output end of simulation information

D.2.2 Node 0 - LNKORD and ASSIGN

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive number of boosters per cluster and time before launch

Receive number of mirrors per orbit, and number of mirror orbits

loop over number of mirror orbits

  loop over number of mirrors per orbit

    initialize mirror utilization arrays

  end loop

end loop

Receive number of lasers

loop over number of lasers

  initial laser utilization array

end loop

Receive simulation time information

Receive time of cluster burnout

START SIMULATION LOOP

  Receive booster position for this time interval

  If not END OF TIME INTERVAL then

    Receive unsorted link calculations from node 1 (3 messages)

```
            Call LNKORD
            Call ASSIGN
            if weapon assignment made then
                  update laser and mirror utilization arrays and number of boosters
                        remaining in cluster
                  Send assignment to HOST
            end if
      else
            Receive message for end of simulation interval
            Send updated utilization information and number of remaining boosters
                  per cluster to node 1 and the host (2 messages each)
            Increment simulation time
      end if
END SIMULATION LOOP
```

D.2.3 Node 1 - RRBVIS, RRPVIS, MIRVIS, & LNKCAL

```
Declare local variables and message types
Open communication channels
Receive number of clusters
Receive number of boosters per cluster and average separation
Receive number of mirrors per orbit, and number of mirror orbits
Receive mirror angular slew acceleration and time to stablize
LOOP over number of mirror orbits
      LOOP over number of mirrors per orbit
            initialize mirror utilization arrays
      end loop
end loop
```

```
Receive number of lasers
Receive laser parameters
LOOP over number of lasers
    initialize laser utilization arrays and laser position vectors
end loop
Receive simulation time information and defenses flag
Receive time of burnout from node 2
START SIMULATION LOOP
    Receive mirror positions for this time interval
    Receive cluster position vector for this time interval
    Receive cluster velocity vector for this time interval
    If defenses on then
        LOOP over number of lasers
            Call RRPVIS
        end loop
        LOOP over booster clusters
            if boosters remain in cluster, before burn out, and
                above minimum altitude then
                Call RRBVIS
                LOOP over number of lasers
                    if laser is available in this time interval
                        Call MIRVIS
                        Call LNKCAL
                end loop
                Send unsorted link data to node 0 (3 messages)
        end loop
    end if
    Send END OF TIME INTERVAL message to nodes 0, 2, and 3
```

Receive updated utilization information and number of remaining boosters per cluster from node 0 (2 messages)

Increment simulation time

END SIMULATION LOOP

D.2.4 Node 2 - BOSTIT and TRAJ

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive cluster types

Receive cluster launch and target positions, re-entry angle, and launch time

LOOP over number of clusters

Call BOSTIT

end loop

Receive simulation time information and defenses flag

START SIMULATION LOOP

LOOP over number of clusters

Determine cluster position and velocity vectors for this time interval

end loop

If not first time interval then wait for END OF TIME INTERVAL message

Send cluster position and velocity vectors to nodes 0, 1, and the host

Increment simulation time

END SIMULATION LOOP

D.2.5 Node 3 - SBMIT and SBMPOS

Declare local variables and message types

Open communication channels

Receive number of mirrors per orbit, and number of mirror orbits

Receive mirror altitude, true anomaly, and orbit inclination

Call SBMIT to initialize mirror orbits

Receive simulation time information

START SIMULATION LOOP

    Call SBMPOS

    If not first time interval then wait for END OF TIME INTERVAL message

    Send mirror positions to node 1 and HOST

    Increment simulation time

END SIMULATION LOOP

D.3 iPSC/1 Implementation #3

D.3.1 Host Program

Declare local variables and message types

Load ALL nodes with link calculation node (MIRVIS, LNKCAL, and LNKORD)

Kill processes in nodes 0, 1, 2, 3, and 4

Load node 0 program (assignment node - ASSIGN)

Load node 1 program (mirror position node - SBMIT and SBMPOS)

Load node 2 program (laser-mirror node - RRPVIS)

Load node 3 program (cluster position node - BOSTIT and TRAJ)

Load node 4 program (cluster-mirror node - RRBVIS)

Open communication channel to nodes

Initialize graphics if Sun workstation host

Read number of clusters

    Swap byte order if Sun workstation host

    Send number of clusters to nodes 0, 3, 4, and

        5 through last node in current cube

Loop over number of clusters

Read cluster information (see implementation #1)

end loop

Swap byte order if Sun workstation host

Send cluster type, launch and target positions, and reentry angle to node 3

Send average separation and number of boosters per cluster to node 4 and nodes 5 through last node in current cube

Send time before launch and number of boosters per cluster to node 0

Read mirror information (see implementation #1)

Swap byte order if Sun workstation host

Send all mirror information to nodes 1 and 5 through last node in current cube

Send number of orbits and mirrors per orbit to node 0, 2, and 4

Read number of lasers

Swap byte order if Sun workstation host

Send number of lasers to nodes 0, 2, and 5 through last node in current cube

Loop over number of lasers

Read laser latitude and longitude

end loop

Read laser parameters (see implementation #1)

Swap byte order if Sun workstation host

Send laser parameters to nodes 2 and 5 through last node in current cube

Read Simulation increment, maximum simulation time, and flag to turn on defenses

Swap byte order if Sun workstation host

Send increment, max time and flag to all nodes

START SIMULATION TIME LOOP

Receive pending message

Swap byte order if Sun workstation host

Case message type

MIMPACT – then received impact time and launch position vector for clusters, used in graphics routines.

MRPOSMSG – received mirror positions, plot if on Sun

MBUFMSG – received cluster positions, plot if on Sun

MSGZAP – received weapon assignment, plot if on Sun

LENGMSG1 – received number of boosters remaining in each cluster at end of simulation increment, increment time

END SIMULATION TIME LOOP

Output end of simulation information

D.3.2 Node 0 – ASSIGN

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive number of boosters per cluster

Receive number of mirrors per orbit, and number of mirror orbits

loop over number of mirror orbits

    loop over number of mirrors per orbit

        initialize mirror utilization arrays

    end loop

end loop

Receive number of lasers

loop over number of lasers

    initial laser utilization array

end loop

```
Receive simulation time information
Receive time of cluster burnout
START SIMULATION LOOP
    Receive booster position for this time interval
    If END OF TIME INTERVAL message not received then
        Receive potential assignment information array
                from nodes 5 through last node in current cube
        LOOP over number of assignments in received array
            Call ASSIGN
            if weapon assignment made then
                update laser and mirror utilization arrays and number of boosters
                    remaining in cluster
                Send assignment to HOST
            end if
        end loop
    else
        Receive message for end of simulation interval
        Send updated utilization information and number of remaining boosters
            per cluster to nodes 2, 4, and 5 through last node in current cube
        Send END OF TIME INTERVAL message to nodes 1 and 3
        Send updated utilization information and number of remaining boosters
            per cluster to the host
        Increment simulation time
    end if
END SIMULATION LOOP
```

D.3.3 Node 1 - SBMIT and SBMPOS

Declare local variables and message types

Open communication channels

Receive number of mirrors per orbit, and number of mirror orbits

Receive mirror altitude, true anomaly, and orbit inclination

Call SBMIT to initialize mirror orbits

Receive simulation time information

START SIMULATION LOOP

    Call SBMPOS

    If not first time interval then wait for END OF TIME INTERVAL message

    Send mirror positions to nodes 2, 4, nodes 5 through the last node

            in the current cube, and the HOST

    Increment simulation time

END SIMULATION LOOP

D.3.4 Node 2 - RRPVIS

Declare local variables and message types

Open communication channels

Receive number of mirrors per orbit, and number of mirror orbits

LOOP over number of mirror orbits

    LOOP over number of mirrors per orbit

        initialize mirror utilization arrays

    end loop

end loop

Receive number of lasers

Receive laser parameters

LOOP over number of lasers

    initialize laser utilization arrays and laser position vectors

end loop

Receive simulation time information and defenses flag

START SIMULATION LOOP

Receive mirror positions for this time interval

if defenses are on then

LOOP over number of lasers

Call RRPVIS

end loop

Send laser-mirror ranges and angles to nodes 5 through

the maximum node id (6 messages)

end if

Receive updated mirror utilization times from node 0

Increment simulation time

END SIMULATION LOOP

D.3.5 Node 3 - BOSTIT and TRAJ

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive cluster types

Receive cluster launch and target positions, re-entry angle, and launch time

LOOP over number of clusters

Call BOSTIT

end loop

Receive simulation time information and defenses flag

START SIMULATION LOOP

LOOP over number of clusters

```
        Determine cluster position and velocity vectors for this time interval
    end loop
    If not first time interval then wait for END OF TIME INTERVAL message
    Send cluster position and velocity vectors to nodes 0, 4, nodes 5 through
            the last node in the current cube, and the host
    Increment simulation time
END SIMULATION LOOP
```

D.3.6 Node 4 - RRBVIS

```
Declare local variables and message types
Open communication channels
Receive number of clusters
Receive number of boosters per cluster and average separation
Receive number of mirrors per orbit, and number of mirror orbits
LOOP over number of mirror orbits
    LOOP over number of mirrors per orbit
        initialize mirror utilization arrays
    end loop
end loop
Receive simulation time information and defenses flag
Receive time of cluster burnout from node 3
START SIMULATION LOOP
    Receive mirror positions for this time interval
    Receive cluster positions for this time interval
    If defenses on then
        LOOP over booster clusters
            if boosters remain in cluster, before burn out, and
                above minimum altitude then
```

Call RRBVIS

Save RRBM, RIANG, and SLANG in vector for next node in loop from node 5 to last node in current cube

if information for 4 clusters has been saved in the vector

Send RRBM, RIANG, and SLANG to that node

if next node is node 5 wait to receive message that node 5 is ready for next cluster

end if

end loop

end if

Send any saved but not sent cluster vectors to the appropriate nodes

Send END OF TIME INTERVAL message to node 5

Receive updated utilization information and number of remaining boosters per cluster from node 0

Increment simulation time

END SIMULATION LOOP

D.3.7 Node 5+ - MIRVIS, LNKCAL, & LNKORD

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive number of boosters per cluster and average separation

Receive number of mirrors per orbit, and number of mirror orbits

Receive mirror angular slew acceleration and time to stablize

Receive number of lasers

Receive laser parameters

LOOP over number of lasers

```
        initialize laser utilization arrays and laser position vectors
end loop
Receive simulation time information and defenses flag
START SIMULATION LOOP
        Receive mirror positions for this time interval
        Receive cluster position and velocity vectors for this time interval
        Receive laser-mirror ranges and angles from node 2 (6 messages)
        If END OF TIME INTERVAL message not yet received then
                Receive cluster vector from node 4
                LOOP over number of clusters in received vector
                        LOOP over number of lasers
                                if laser is available in this time interval
                                        Call MIRVIS
                                        Call LNKCAL
                        end loop
                        Call LNKORD
                        Save "NMLIK" potential assignments in array for node 0
                        if array for node 0 is full then
                                Send array of potential assignments to node 0
                        end if
                end loop
        end if
        Send any non-empty arrays of potential assignments to node 0
        Receive END OF TIME INTERVAL message
        if not last node in current cube then
                Send END OF TIME INTERVAL message to next node in
                range 5 to last node
        else
```

Send END OF TIME INTERVAL message to node 0

end if

Receive updated utilization information and number of remaining boosters per cluster from node 0 (2 messages)

Increment simulation time

END SIMULATION LOOP

D.4 iPSC/1 Implementation #4

D.4.1 Host Program

Declare local variables and message types

Load ALL nodes with link calculation node (MIRVIS, LNKCAL, and LNKORD)

Kill processes in nodes 0, 1, 2, and 3

Load node 0 program (assignment node - ASSIGN)

Load node 1 program (mirror position node - SBMIT and SBMPOS)

Load node 2 program (cluster position node - BOSTIT and TRAJ)

Load node 3 program (cluster-mirror node - RRBVIS)

Open communication channel to nodes

Initialize graphics if Sun workstation host

Read number of clusters

Swap byte order if Sun workstation host

Send number of clusters to nodes 0, 2, 3, and 4 through last node in current cube

Loop over number of clusters

Read cluster information (see implementation #1)

end loop

Swap byte order if Sun workstation host

Send cluster type, launch and target positions, and reentry angle to node 2

Send average separation and number of boosters per cluster to

node 3 and nodes 4 through last node in current cube

Send time before launch and number of boosters per cluster to node 0

Read mirror information (see implementation #1)

Swap byte order if Sun workstation host

Send all mirror information to nodes 1 and 4 through last node in current cube

Send number of orbits and mirrors per orbit to node 0 and 3

Read number of lasers

Swap byte order if Sun workstation host

Send number of lasers to nodes 0 and 4 through last node in current cube

Loop over number of lasers

Read laser latitude and longitude

end loop

Read laser parameters (see implementation #1)

Swap byte order if Sun workstation host

Send laser parameters to nodes 4 through last node in cube

Read Simulation increment, maximum simulation time, and flag

to turn on defenses

Swap byte order if Sun workstation host

Send increment, max time and flag to all nodes

START SIMULATION TIME LOOP

Receive pending message

Swap byte order if Sun workstation host

Case message type

MIMPACT - then received impact time and launch position

vector for clusters, used in graphics routines.

MRPOSMSG - received mirror positions, plot if on Sun

MBUFMSG – received cluster positions, plot if on Sun

MSGZAP – received weapon assignment, plot if on Sun

LENGMSG1 – received number of boosters remaining in each

cluster at end of simulation increment, increment time

END SIMULATION TIME LOOP

Output end of simulation information

D.4.2 Node 0 - ASSIGN

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive number of boosters per cluster

Receive number of mirrors per orbit, and number of mirror orbits

loop over number of mirror orbits

    loop over number of mirrors per orbit

        initialize mirror utilization arrays

    end loop

end loop

Receive number of lasers

loop over number of lasers

    initial laser utilization array

end loop

Receive simulation time information

Receive time of cluster burnout

START SIMULATION LOOP

    Receive booster position for this time interval

    If END OF TIME INTERVAL message not received then

```
            Receive potential assignment information array
                    from nodes 5 through last node in current cube
            LOOP over number of assignments in received array
                Call ASSIGN
                if weapon assignment made then
                    update laser and mirror utilization arrays and number of boosters
                        remaining in cluster
                    Send assignment to HOST
                end if
            end loop
        else
            Receive END OF TIME INTERVAL message
            Send updated utilization information and number of remaining boosters
                per cluster to nodes 2, 4, and 5 through last node in current cube
            Send END OF TIME INTERVAL message to nodes 1 and 2
            Send updated utilization information and number of remaining boosters
                per cluster to the host
            Increment simulation time
        end if
    END SIMULATION LOOP
```

D.4.3 Node 1 - SBMIT and SBMPOS

```
Declare local variables and message types
Open communication channels
Receive number of mirrors per orbit, and number of mirror orbits
Receive mirror altitude, true anomaly, and orbit inclination
Call SBMIT to initialize mirror orbits
```

Receive simulation time information

START SIMULATION LOOP

Call SBMPOS

If not first time interval then wait for END OF TIME INTERVAL message

Send mirror positions to nodes 3 through the last node in the current cube, and the HOST

Increment simulation time

END SIMULATION LOOP

D.4.4 Node 2 - BOSTIT and TRAJ

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive cluster types

Receive cluster launch and target positions, re-entry angle, and launch time

LOOP over number of clusters

Call BOSTIT

end loop

Receive simulation time information and defenses flag

START SIMULATION LOOP

LOOP over number of clusters

Determine cluster position and velocity vectors for this time interval

end loop

If not first time interval then wait for END OF TIME INTERVAL message

Send cluster position and velocity vectors to nodes 0, 3, nodes 4 through the last node in the current cube, and the host

Increment simulation time

END SIMULATION LOOP

D.4.5 Node 3 - RRBVIS

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive number of boosters per cluster and average separation

Receive number of mirrors per orbit, and number of mirror orbits

LOOP over number of mirror orbits

  LOOP over number of mirrors per orbit

    initialize mirror utilization arrays

  end loop

end loop

Receive simulation time information and defenses flag

Receive time of cluster burnout from node 2

START SIMULATION LOOP

  Receive mirror positions for this time interval

  Receive cluster positions for this time interval

  If defenses on then

    LOOP over booster clusters

      If boosters remain in cluster, before burn out, and

        above minimum altitude then

        Call RRBVIS

        Save RRBM, RIANG, and SLANG in vector for next node in loop from

          node 4 to last node in current cube

        if information for 4 clusters has been saved in the vector

          Send RRBM, RIANG, and SLANG to that node

          if next node is node 4 wait to receive message that node 4 is ready for next cluster

end if

end loop

end if

Send any saved but not sent cluster vectors to the appropriate nodes

Send END OF TIME INTERVAL message to node 4

Receive updated utilization information and number of remaining boosters per cluster from node 0

Increment simulation time

END SIMULATION LOOP

D.4.6 Node 4+ – RRPVIS, MIRVIS, LNKCAL, & LNKORD

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive number of boosters per cluster and average separation

Receive number of mirrors per orbit, and number of mirror orbits

Receive mirror angular slew acceleration and time to stablize

Receive number of lasers

Receive laser parameters

LOOP over number of lasers

initialize laser utilization arrays and laser position vectors

end loop

Receive simulation time information and defenses flag

START SIMULATION LOOP

Receive mirror positions for this time interval

Receive cluster position and velocity vectors for this time interval

```
LOOP over number of lasers
    if laser is available this time interval Call RRPVIS
end loop
If END OF TIME INTERVAL message not yet received then
    Receive cluster vector from node 4
    LOOP over number of clusters in received vector
        LOOP over number of lasers
            if laser is available in this time interval
                Call MIRVIS
                Call LNKCAL
        end loop
        Call LNKORD
        Save "NMLIK" potential assignments in array for node 0
        if array for node 0 is full then
            Send array of potential assignments to node 0
        end if
    end loop
end if
Send any non-empty arrays of potential assignments to node 0
Receive END OF TIME INTERVAL message
if not last node in current cube then
    Send END OF TIME INTERVAL message to next node in
    range 4 to last node
else
    Send END OF TIME INTERVAL message to node 0
end if
Receive updated utilization information and number of remaining boosters
    per cluster from node 0 (2 messages)
```

Increment simulation time

END SIMULATION LOOP

D.5 iPSC/1 Implementation #5

D.5.1 Host Program

Declare local variables and message types

Load ALL nodes with link calculation node (MIRVIS, LNKCAL, and LNKORD)

Kill processes in nodes 0, 1, 2, and 3

Load node 0 program (assignment node - ASSIGN)

Load node 1 program (mirror position node - SBMIT and SBMPOS)

Load node 2 program (cluster position node - BOSTIT and TRAJ)

Load node 3 program (Supervisor node)

Open communication channel to nodes

Initialize graphics if Sun workstation host

Read number of clusters

    Swap byte order if Sun workstation host

    Send number of clusters to nodes 0, 2, 3, and

        4 through last node in current cube

Loop over number of clusters

        Read cluster information (see implementation #1)

end loop

    Swap byte order if Sun workstation host

        Send cluster type, launch and target positions, and reentry angle

            to node 2

        Send average separation and number of boosters per cluster to

            node 3 and nodes 4 through last node in current cube

        Send time before launch and number of boosters per cluster to node 0

Read mirror information (see implementation #1)

  Swap byte order if Sun workstation host

  Send all mirror information to nodes 1 and 4 through last node in current cube

  Send number of orbits and mirrors per orbit to node 0 and 3

Read number of lasers

  Swap byte order if Sun workstation host

  Send number of lasers to nodes 0 and 4 through last node in current cube

Loop over number of lasers

  Read laser latitude and longitude

end loop

Read laser parameters (see implementation #1)

  Swap byte order if Sun workstation host

  Send laser parameters to nodes 4 through last node in cube

Read Simulation increment, maximum simulation time, and flag

  to turn on defenses

  Swap byte order if Sun workstation host

  Send increment, max time and flag to all nodes

START SIMULATION TIME LOOP

  Receive pending message

  Swap byte order if Sun workstation host

  Case message type

    MIMPACT - then received impact time and launch position

      vector for clusters, used in graphics routines.

    MRPOSMSG - received mirror positions, plot if on Sun

    MBUFMSG - received cluster positions, plot if on Sun

    MSGZAP - received weapon assignment, plot if on Sun

    LENGMSG1 - received number of boosters remaining in each

      cluster at end of simulation increment, increment time

END SIMULATION TIME LOOP

Output end of simulation information

D.5.2 Node 0 - ASSIGN

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive number of boosters per cluster

Receive number of mirrors per orbit, and number of mirror orbits

loop over number of mirror orbits

    loop over number of mirrors per orbit

        initialize mirror utilization arrays

    end loop

end loop

Receive number of lasers

loop over number of lasers

    initial laser utilization array

end loop

Receive simulation time information

Receive time of cluster burnout

START SIMULATION LOOP

    Receive booster position for this time interval

    If END OF TIME INTERVAL message not received then

        Receive potential assignment information array

                from nodes 5 through last node in current cube

        LOOP over number of assignments in received array

            Call ASSIGN

if weapon assignment made then

update laser and mirror utilization arrays and number of boosters

remaining in cluster

Send assignment to HOST

end if

end loop

else

Receive END OF TIME INTERVAL message

Send updated utilization information and number of remaining boosters

per cluster to nodes 2, 4, and 5 through last node in current cube

Send END OF TIME INTERVAL message to nodes 1 and 2

Send updated utilization information and number of remaining boosters

per cluster to the host

Increment simulation time

end if

END SIMULATION LOOP

D.5.3 Node 1 - SBMIT and SBMPOS

Declare local variables and message types

Open communication channels

Receive number of mirrors per orbit, and number of mirror orbits

Receive mirror altitude, true anomaly, and orbit inclination

Call SBMIT to initialize mirror orbits

Receive simulation time information

START SIMULATION LOOP

Call SBMPOS

If not first time interval then wait for END OF TIME INTERVAL message

Send mirror positions to nodes 4 through the last node in the current cube, and the HOST

Increment simulation time

END SIMULATION LOOP

D.5.4 Node 2 - BOSTIT and TRAJ

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive cluster types

Receive cluster launch and target positions, re-entry angle, and launch time

LOOP over number of clusters

Call BOSTIT

end loop

Receive simulation time information and defenses flag

START SIMULATION LOOP

LOOP over number of clusters

Determine cluster position and velocity vectors for this time interval

end loop

If not first time interval then wait for END OF TIME INTERVAL message

Send cluster position and velocity vectors to nodes 0, 3, nodes 4 through the last node in the current cube, and the host

Increment simulation time

END SIMULATION LOOP

D.5.5 Node 3 - Supervisor node

Declare local variables and message types

```
Open communication channels
Receive number of clusters
Receive number of boosters per cluster and average separation
Receive number of mirrors per orbit, and number of mirror orbits
LOOP over number of mirror orbits
    LOOP over number of mirrors per orbit
        initialize mirror utilization arrays
    end loop
end loop
Receive simulation time information and defenses flag
Receive time of cluster burnout from node 2
START SIMULATION LOOP
    Receive cluster positions for this time interval
    If defenses on then
        LOOP over booster clusters
            if boosters remain in cluster, before burn out, and
                above minimum altitude then
                Save cluster index in vector for next node in the range
                    node 4 to last node in current cube
                if information for 4 clusters has been saved in the vector
                    Send the vector to that node
                end if
        end loop
    end if
    Send any saved but not sent cluster vectors to the appropriate nodes
    Send END OF TIME INTERVAL message to node 4
    Receive updated utilization information and number of remaining boosters
        per cluster from node 0
```

Increment simulation time

END SIMULATION LOOP

D.5.6 Node 4+ - RRPVIS, RRBVIS, MIRVIS, LNKCAL, & LNKORD

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive number of boosters per cluster and average separation

Receive number of mirrors per orbit, and number of mirror orbits

Receive mirror angular slew acceleration and time to stablize

Receive number of lasers

Receive laser parameters

LOOP over number of lasers

    initialize laser utilization arrays and laser position vectors

end loop

Receive simulation time information and defenses flag

START SIMULATION LOOP

    Receive mirror positions for this time interval

    Receive cluster position and velocity vectors for this time interval

    LOOP over number of lasers

        if laser is available this time interval Call RRPVIS

    end loop

    If END OF TIME INTERVAL message not yet received then

        Receive cluster vector from node 4

        LOOP over number of clusters in received vector

            Call RRBVIS

            LOOP over number of lasers

```
                if laser is available in this time interval
                    Call MIRVIS
                    Call LNKCAL
            end loop
            Call LNKORD
            Save "NMLIK" potential assignments in array for node 0
            if array for node 0 is full then
                Send array of potential assignments to node 0
            end if
        end loop
    end if
    Send any non-empty arrays of potential assignments to node 0
    Receive END OF TIME INTERVAL message
    if not last node in current cube then
        Send END OF TIME INTERVAL message to next node in
        range 4 to last node
    else
        Send END OF TIME INTERVAL message to node 0
    end if
    Receive updated utilization information and number of remaining boosters
        per cluster from node 0
    Increment simulation time
END SIMULATION LOOP
```

D.6 iPSC/1 Implementation #6

D.6.1 Host Program

Declare local variables and message types

```
Load ALL nodes with link calculation node (MIRVIS, LNKCAL, and LNKORD)
Kill processes in nodes 0 and 1
Load node 0 program (assignment node - ASSIGN)
Load node 1 program (Supervisor node)
Open communication channel to nodes
Initialize graphics if Sun workstation host
Read number of clusters
    Swap byte order if Sun workstation host
    Send number of clusters to all nodes
Loop over number of clusters
        Read cluster information (see implementation #1)
end loop
    Swap byte order if Sun workstation host
        Send cluster type, launch and target positions, and reentry angle
            to nodes 2 through the last node in the current cube
        Send average separation and number of boosters per cluster to
            nodes 2 through last node in current cube
        Send time before launch and number of boosters per cluster to node 0
Read mirror information (see implementation #1)
    Swap byte order if Sun workstation host
    Send all mirror information to nodes 2 through last node in current cube
    Send number of orbits and mirrors per orbit to node 0
Read number of lasers
    Swap byte order if Sun workstation host
    Send number of lasers to nodes 0 and 2 through last node in current cube
Loop over number of lasers
    Read laser latitude and longitude
end loop
```

Read laser parameters (see implementation #1)

Swap byte order if Sun workstation host

Send laser parameters to nodes 2 through last node in cube

Read Simulation increment, maximum simulation time, and flag to turn on defenses

Swap byte order if Sun workstation host

Send increment, max time and flag to all nodes

START SIMULATION TIME LOOP

Receive pending message

Swap byte order if Sun workstation host

Case message type

MIMPACT - then received impact time and launch position vector for clusters, used in graphics routines.

MRPOSMSG - received mirror positions, plot if on Sun

MBUFMSG - received cluster positions, plot if on Sun

MSGZAP - received weapon assignment, plot if on Sun

LENGMSG1 - received number of boosters remaining in each cluster at end of simulation increment, increment time

END SIMULATION TIME LOOP

Output end of simulation information

D.6.2 Node 0 - ASSIGN

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive number of boosters per cluster

Receive number of mirrors per orbit, and number of mirror orbits

```
loop over number of mirror orbits
    loop over number of mirrors per orbit
        initialize mirror utilization arrays
    end loop
end loop
Receive number of lasers
loop over number of lasers
    initial laser utilization array
end loop
Receive simulation time information
Receive time of cluster burnout
START SIMULATION LOOP
    Receive booster position for this time interval
    If END OF TIME INTERVAL message not received then
        Receive potential assignment information array
                from nodes 2 through last node in current cube
        LOOP over number of assignments in received array
            Call ASSIGN
            if weapon assignment made then
                update laser and mirror utilization arrays and number of boosters
                    remaining in cluster
                Send assignment to HOST
            end if
        end loop
    else
        Receive END OF TIME INTERVAL message
        Send updated utilization information and number of remaining boosters
            per cluster to nodes 1 through last node in current cube and the host
```

```
        Increment simulation time
    end if
END SIMULATION LOOP
```

D.6.3 Node 1 - Supervisor node

```
Declare local variables and message types
Open communication channels
Receive number of clusters
Receive number of boosters per cluster and average separation
Receive number of mirrors per orbit, and number of mirror orbits
LOOP over number of mirror orbits
    LOOP over number of mirrors per orbit
        initialize mirror utilization arrays
    end loop
end loop
Receive simulation time information and defenses flag
Receive time of cluster burnout from node 2
START SIMULATION LOOP
    Receive cluster positions for this time interval
    If defenses on then
        LOOP over booster clusters
            if boosters remain in cluster, before burn out, and
                above minimum altitude then
                Save cluster index in vector for next node in the range
                    node 2 to last node in current cube
                if information for 4 clusters has been saved in the vector
                    Send the vector to that node
```

```
                end if
            end loop
        end if
        Send any saved but not sent cluster vectors to the appropriate nodes
        Send END OF TIME INTERVAL message to node 2
        Receive updated utilization information and number of remaining boosters
            per cluster from node 0
        Increment simulation time
    END SIMULATION LOOP
```

D.6.4 Node 2+ - SBMIT, SBMPOS, BOSTIT, TRAJ, RRPVIS, RRBVIS, MIRVIS, LNKCAL, & LNKORD

```
Declare local variables and message types
Open communication channels
Receive number of clusters
Receive cluster types
Receive cluster launch and target positions, re-entry angle, and launch time
LOOP over number of clusters
    Call BOSTIT
end loop
Receive number of boosters per cluster and average separation
Receive number of mirrors per orbit, and number of mirror orbits
Receive initialization data
Call SBMIT
Receive number of lasers
Receive laser parameters
LOOP over number of lasers
```

```
        initialize laser utilization arrays and laser position vectors
end loop
Receive simulation time information and defenses flag
Send cluster impact times to host
Send time to cluster burnout to nodes 0 and 1
START SIMULATION LOOP
        Call SBMPOS
        if I am node 2 Send mirror positions to the host
        LOOP over number of clusters
                Call TRAJ
        end loop
        if I am node 2 Send booster position and velocity vectors
                to nodes 0 and 1, and the host
        LOOP over number of lasers
                if laser is available this time interval Call RRPVIS
        end loop
        If END OF TIME INTERVAL message not yet received then
                Receive cluster vector from node 1
                LOOP over number of clusters in received vector
                        Call RRBVIS
                        LOOP over number of lasers
                                if laser is available in this time interval
                                        Call MIRVIS
                                        Call LNKCAL
                        end loop
                        Call LNKORD
                        Save "NMLIK" potential assignments in array for node 0
                        if array for node 0 is full then
                                Send array of potential assignments to node 0
```

end if

end loop

end if

Send any non-empty arrays of potential assignments to node 0

Receive END OF TIME INTERVAL message

if not last node in current cube then

Send END OF TIME INTERVAL message to next node in range 2 to last node

else

Send END OF TIME INTERVAL message to node 0

end if

Receive updated utilization information and number of remaining boosters per cluster from node 0

Increment simulation time

END SIMULATION LOOP

D.7 iPSC/1 Implementation #7

D.7.1 Host Program

Declare local variables and message types

Load ALL nodes with link calculation node (MIRVIS, LNKCAL, & LNKORD)

Kill processes in nodes 0 and 1

Load node 0 program (assignment node - ASSIGN)

Load node 1 program (Supervisor node)

Open communication channel to nodes

Initialize graphics if Sun workstation host

Read number of clusters

Swap byte order if Sun workstation host

```
          Send number of clusters to all nodes
Loop over number of clusters
                    Read cluster information (see implementation #1)
end loop
          Swap byte order if Sun workstation host
                    Send cluster type, launch and target positions, and reentry angle
                              to all nodes
                    Send average separation and number of boosters per cluster to
                              nodes 2 through last node in current cube
                    Send time before launch and number of boosters per cluster to node 0
Read mirror information (see implementation #1)
          Swap byte order if Sun workstation host
          Send all mirror information to nodes 0 and 2 through last node in current cube
Read number of lasers
          Swap byte order if Sun workstation host
          Send number of lasers to nodes 0 and 2 through last node in current cube
Loop over number of lasers
          Read laser latitude and longitude
end loop
Read laser parameters (see implementation #1)
          Swap byte order if Sun workstation host
          Send laser parameters to nodes 2 through last node in cube
Read Simulation increment, maximum simulation time, and flag
          to turn on defenses
          Swap byte order if Sun workstation host
          Send increment, max time and flag to all nodes
START SIMULATION TIME LOOP
          Receive pending message
```

Swap byte order if Sun workstation host

Case message type

MIMPACT – then received impact time and launch position vector for clusters, used in graphics routines.

MRPOSMSG – received mirror positions, plot if on Sun

MBUFMSG – received cluster positions, plot if on Sun

MSGZAP – received weapon assignment, plot if on Sun

LENGMSG1 – received number of boosters remaining in each cluster at end of simulation increment, increment time

END SIMULATION TIME LOOP

Output end of simulation information

D.7.2 Node 0 - SBMIT, SBMPOS, BOSTIT, TRAJ, & ASSIGN

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive number of boosters per cluster

Receive cluster types

Receive cluster launch and target positions, re-entry angle, and launch time

LOOP over number of clusters

Call BOSTIT

end loop

Receive number of mirrors per orbit, and number of mirror orbits

Receive mirror initialization data

Call SBMIT

LOOP over number of mirror orbits

LOOP over number of mirrors per orbit

```
          initialize mirror utilization arrays
     end loop
end loop
Receive number of lasers
LOOP over number of lasers
     initial laser utilization array
end loop
Receive simulation time information
Send impact times to host
START SIMULATION LOOP
     Call SBMPOS
     Send mirror positions to host
     LOOP over number of clusters
          Call TRAJ end loop
     Send cluster positions to host
     If END OF TIME INTERVAL message not received then
          Receive potential assignment information array
                    from nodes 2 through last node in current cube
          LOOP over number of assignments in received array
               Call ASSIGN
               if weapon assignment made then
                    update laser and mirror utilization arrays and number of boosters
                         remaining in cluster
                    Send assignment to HOST
               end if
          end loop
     else
          Receive END OF TIME INTERVAL message
```

```
        Send updated utilization information and number of remaining boosters
            per cluster to all nodes and the host
        Increment simulation time
    end if
END SIMULATION LOOP
```

D.7.3 Node 1 - BOSTIT, TRAJ, Supervisor node

```
Declare local variables and message types
Open communication channels
Receive number of clusters
Receive cluster types
Receive cluster launch and target positions, re-entry angle, and launch time
LOOP over number of clusters
    Call BOSTIT
end loop
Receive number of boosters per cluster and average separation
Receive simulation time information and defenses flag
START SIMULATION LOOP
    if first time through simulation loop then
        LOOP over number of clusters
            Call TRAJ
        end loop
    end if
    If defenses on then
        LOOP over booster clusters
            if boosters remain in cluster, before burn out, and
                above minimum altitude then
```

Save cluster index in vector for next node in the range

node 2 to last node in current cube

end if

end loop

end if

LOOP over range 2 to last node in current cube

Send cluster vector to the appropriate node

end loop

Send END OF TIME INTERVAL message to node 2

Increment simulation time

if not last time through loop then

LOOP over number of clusters

Call TRAJ with new time

end loop

end if

Receive updated utilization information and number of remaining boosters

per cluster from node 0

END SIMULATION LOOP

D.7.4 Node 2+ - SBMIT, SBMPOS, BOSTIT, TRAJ, RRPVIS, RRBVIS, MIRVIS, LNKCAL, & LNKORD

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive number of boosters per cluster and average separation

Receive number of mirrors per orbit, and number of mirror orbits

Receive mirror angular slew acceleration and time to stablize

```
Receive number of lasers
Receive laser parameters
LOOP over number of lasers
    initialize laser utilization arrays and laser position vectors
end loop
Receive simulation time information and defenses flag
Call SBMPOS
START SIMULATION LOOP
    LOOP over number of lasers
        if laser is available this time interval Call RRPVIS
    end loop
    If END OF TIME INTERVAL message not yet received then
        Receive cluster vector from node 1
        LOOP over number of clusters in received vector
            Call TRAJ
            Call RRBVIS
            LOOP over number of lasers
                if laser is available in this time interval
                    Call MIRVIS
                    Call LNKCAL
            end loop
            Call LNKORD
            Save "NMLIK" potential assignments in array for node 0
            if array for node 0 is full then
                Send array of potential assignments to node 0
            end if
        end loop
    end if
```

Send any non-empty arrays of potential assignments to node 0

Receive END OF TIME INTERVAL message

if not last node in current cube then

Send END OF TIME INTERVAL message to next node in range 4 to last node

else

Send END OF TIME INTERVAL message to node 0

end if

Call SBMPOS for next time

Receive updated utilization information and number of remaining boosters per cluster from node 0

Increment simulation time

END SIMULATION LOOP

D.8 iPSC/1 Implementation #7

D.8.1 Host Program

Declare local variables and message types

Load ALL nodes with link calculation node (MIRVIS, LNKCAL, and LNKORD)

Kill processes in node 0

Load node 0 program (assignment node - ASSIGN)

Open communication channel to nodes

Initialize graphics if Sun workstation host

Read number of clusters

Swap byte order if Sun workstation host

Send number of clusters to all nodes

Loop over number of clusters

Read cluster information (see implementation #1)

end loop

Swap byte order if Sun workstation host

Send cluster type, launch and target positions, and reentry angle to all nodes

Send average separation and number of boosters per cluster to nodes 1 through last node in current cube

Send time before launch and number of boosters per cluster to node 0

Read mirror information (see implementation #1)

Swap byte order if Sun workstation host

Send all mirror information to all nodes

Read number of lasers

Swap byte order if Sun workstation host

Send number of lasers to all nodes

Loop over number of lasers

Read laser latitude and longitude

end loop

Read laser parameters (see implementation #1)

Swap byte order if Sun workstation host

Send laser parameters to nodes 1 through last node in cube

Read Simulation increment, maximum simulation time, and flag to turn on defenses

Swap byte order if Sun workstation host

Send increment, max time and flag to all nodes

START SIMULATION TIME LOOP

Receive pending message

Swap byte order if Sun workstation host

Case message type

MIMPACT - then received impact time and launch position vector for clusters, used in graphics routines.

MRPOSMSG - received mirror positions, plot if on Sun

MBUFMSG - received cluster positions, plot if on Sun

MSGZAP - received weapon assignment, plot if on Sun

LENGMSG1 - received number of boosters remaining in each cluster at end of simulation increment, increment time

END SIMULATION TIME LOOP

Output end of simulation information

D.8.2 Node 0 - ASSIGN

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive number of boosters per cluster

Receive cluster types

Receive cluster launch and target positions, re-entry angle, and launch time

LOOP over number of clusters

Call BOSTIT

end loop

Receive number of mirrors per orbit, and number of mirror orbits

Receive mirror initialization data

Call SBMIT

LOOP over number of mirror orbits

LOOP over number of mirrors per orbit

initialize mirror utilization arrays

end loop

end loop

```
Receive number of lasers
LOOP over number of lasers
    initial laser utilization array
end loop
Receive simulation time information
Send impact times to host
START SIMULATION LOOP
    Call SBMPOS
    Send mirror positions to host
    LOOP over number of clusters
        Call TERAL
    end loop
    Send cluster positions to host
    If END OF TIME INTERVAL message not received then
        Receive potential assignment information array
                from nodes 2 through last node in current cube
        LOOP over number of assignments in received array
            Call ASSIGN
            if weapon assignment made then
                update laser and mirror utilization arrays and number of boosters
                    remaining in cluster
                Send assignment to HOST
            end if
        end loop
    else
        Receive END OF TIME INTERVAL message
        Send updated utilization information and number of remaining boosters
            per cluster to all nodes and the host
        Increment simulation time
```

end if

END SIMULATION LOOP

D.8.3 Node 1+ - RRPVIS, RRBVIS, MIRVIS, LNKCAL, & LNKORD

Declare local variables and message types

Open communication channels

Receive number of clusters

Receive number of boosters per cluster and average separation

Receive number of mirrors per orbit, and number of mirror orbits

Receive mirror angular slew acceleration and time to stablize

Receive number of lasers

Receive laser parameters

LOOP over number of lasers

    initialize laser utilization arrays and laser position vectors

end loop

Receive simulation time information and defenses flag

Call SBMPOS

START SIMULATION LOOP

    LOOP over number of lasers

        if laser is available this time interval Call RRPVIS

    end loop

    If END OF TIME INTERVAL message not yet received then

        LOOP over number of clusters modulo my node number

            Call TRAJ

            Call RRBVIS

            LOOP over number of lasers

                if laser is available in this time interval

```
                Call MIRVIS
                Call LNKCAL
            end loop
            Call LNKORD
            Save "NMLIK" potential assignments in array for node 0
            if array for node 0 is full then
                Send array of potential assignments to node 0
            end if
        end loop
    end if
    Send any non-empty arrays of potential assignments to node 0
    if node 1 then
        Send END OF TIME INTERVAL message to node 2
    else
        Receive END OF TIME INTERVAL message
        if not last node in current cube then
            Send END OF TIME INTERVAL message to next node in
            range 3 to last node
        else
            Send END OF TIME INTERVAL message to node 0
        end if
    end if
    Call SBMPOS to determine mirror positions for next time interval
    Receive updated utilization information and number of remaining boosters
        per cluster from node 0
    Increment simulation time
END SIMULATION LOOP
```

D.9 Encore Implementation

```
Declare all global data structures
Read in booster initialization information
BEGIN PARALLEL
    LOOP over number of clusters
        Call BOSTIT
    end loop
END PARALLEL
Read in mirror initialization information
Call SBMIT
LOOP over number of mirror orbits
    LOOP over number of mirrors per orbit
        initialize mirror utilization arrays
    end loop
end loop
Read in laser initialization information
LOOP over number of lasers
    Determine laser position and initialize utilization times
end loop
Read in simulation times and defense flag
BEGIN SIMULATION LOOP
    Call SBMPOS
    BEGIN PARALLEL
        LOOP over number of clusters
            Call TRAJ
        end loop
    END PARALLEL
    if defenses are on then
```

```
            BEGIN PARALLEL
                LOOP over the number of lasers
                    Call RRPVIS if laser is available in this time interval
                end loop
            END PARALLEL
            BEGIN PARALLEL
            Declare local data structures required
            LOOP over booster clusters
                if cluster is above minimum altitude and before burnout then
                    LOOP over available lasers
                        Call MIRVIS
                        Call LNKCAL
                    end loop
                    Call LNKORD
                    BEGIN CRITICAL SECTION
                        Call ASSIGN
                        Display assignment
                    END CRITICAL SECTION
                end if
            end loop
            END PARALLEL
        end if
        Increment Simulation time
    END SIMULATION LOOP
    Display simulation results
```

Bibliography

1. Banks, Jerry and John S. Carson. *Discrete Event Simulation*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

2. Baum, Alan M. and Donald J. McMillan. "Automated Parallelization of Serial Simulations for Hypercube Parallel Processors." In *Distributed Simulation 1989*, La Jolla CA: SCS, 1989.

3. Beckman, Brian, et al. "Instantaneous Speedup." To be published in summmer simulation conference 1989, April 1989.

4. Beckman, Brian and P. Hontalas and J. Ruffles and F. Wieland and D. Jefferson. "Distributed Simulation and Time Warp, Part1: Desig of *Colliding Pucks*." In Unger, B. and D. Jefferson, editors, *Distributed Simulation 1988*, Volume 19, pages 56–60, La Jolla CA: SCS, February 1988.

5. Biles, William E. "Introduction to Simulation." In *Proceedings of the 1987 Winter Simulation Conference*, pages 7–15, December 1987.

6. Bryant, Randal E. "Simulation on a Distributed System." In *Proceedings of the 1st Int'l Conference on Distributed Computing Systems*, pages 544–552, October 1979.

7. Chandy, K. Mani and Jayadev Misra. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, *5*(5):440–452 (September 1979).

8. Chandy, K. Mani and Rivi Sherman. "The Conditional Event Approach to Distributed Simulation." In *Distributed Simulation 1989*, La Jolla CA: SCS, 1989.

9. Chandy, K.M. and J. Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, *24*(11):19?-206 (April 1981).

10. Cho, C. K., E. K. Lin and C. L. Jen. "On Performance Evaluations of Multiprocessor Systems for Real-time Simulation." In *Proceedings of the 17th Annual Simulation Symposium*, pages 209–225, March 1984.

11. Comfort, John Craig. "The Design of a Multiprocessor Based Simulation Computer - II." In *Proceedings of the Sixteenth Annual Simulation Symposium*, pages 197–209, 1983.

12. DESE Research and Development, Inc., Huntsville, Alabama. *A Method for Improving Technology Research and Development Decisions Regarding BMD and ASAT*, 1985. Volume II - Simulation System Design Guide.

13. Encore. *Multimax Technical Summary*. Encore Computer Corporation, Marlboro, MA, March 1987.

14. Fujimoto, Richard M. "Performance Measurements of Distributed Simulation Strategies." In *Distributed Simulation 1988*, La Jolla CA: SCS, 1988.

15. Gilmer Jr., John B. and Jung Pyo Hong. "Replicated State Space Approach for Parallel Simulation." In Wilson, J., et al., editors, *Proceedings of the 1986 Winter Simulation Conference*, pages 430–434, IEEE, December 1986.

16. Glover, Charles. "Techniques for Converting Sequential Programs Into Concurrent Programs for a Hypercube Computer." draft copy, 1988.

17. Hartrum, Thomas C. and Brian J. Donlan. "Distributed battle-management simulation on a Hypercube." In Unger, B. and D. Jefferson, editors, *Distributed Simulation 1988*, pages 3–7, La Jolla CA: SCS, February 1988.

18. Heidelberger, Philip. "Statistical Analysis of Parallel Simulation." In *Proceedings of the 1986 Winter Simulation Conference*, pages 290–295, 1986.

19. Hoare, C.A.R. "Communicating Sequential Processes," *Communications of the ACM*, *21*(8):666–677 (August 1978).

20. Hwang, Kai and Faye A. Briggs. *Computer Architecture and Parallel Processing*. New York NY: McGraw-Hill, 1984.

21. Intel. *iPSC System Overview Manual*. Intel Scientific Computers, Beaverton, Oregon, November 1986. Order Number 310610-001.

22. Intel. *iPSC/2 User's Guide (Preliminary)*. Intel Scientific Computers, Beaverton, Oregon, March 1988. Order Number 311532-002.

23. Jefferson, David. "Virtual Time," *ACM Transactions on Programming Languages and Systems*, 7(3):404–425 (July 1985).

24. Jefferson, David and Henry Sowizral. "Fast Concurrent Simulation using the Time Warp Mechanism." In *Distributed Simulation 1985*, La Jolla CA: SCS, 1985.

25. Jefferson, David, et al. ""The Status of the Time Warp Operating System"." In *Symposium on Operating Systems Principles*, pages 738–744, ACM, 1988.

26. Jones, Douglas W. "Concurrent Simulation: An Alternative to Distributed Simulation." In *Proceedings of the 1986 Winter Simulation Conference*, pages 417–423, 1986.

27. Kaudel, Fred J. "A Literature Survey on Distributed Discrete Event Simulation," *Simuletter*, *18*(2):11–21 (June 1987).

28. Lin, Eric K. and Chian-Li Jen. "Contention Problem of a Multiprocessor Simulator." In *Proceedings of the 16th Annual Simulation Symposium*, pages 229–238, March 1983.

29 Misra, Jayadev. "Distributed Discrete-Event Simulation," *ACM Computing Surveys*, *18*(1):39–65 (March 1986).

30. Nicol, David M. "Mapping a battlefield simulation onto message-passing parallel architectures." In *Distributed Simulation 1988*, La Jolla CA: SCS, 1988.

31. Nicol, David M. "Dynamic Remapping of Parallel Time-stepped Simulations." In *Distributed Simulation 1989*, La Jolla CA: SCS, 1989.

32. Nicol, David M. and Paul F. Reynolds, Jr. "Problem Oriented Protocol Design." In *Proceedings of the 1984 Winter Simulation Conference*, pages 471–474, November 1984.

33. Peterson, James L. and Abraham Silberschatz. *Operating System Concepts* (Second Edition). Reading, MA: Addison-Wesley, 1985.

34. Pritsker, A. Alan B. *Introduction to Simulation and SLAM II*. West Lafayette IN: Systems Publishing Corp., 1986.

35. Quinn, Michael J. *Designing Efficient Algorithms for Parallel Computers*. New York NY: McGraw-Hill, 1987.

36. Reed, Daniel A. "Parallel Discrete Event Simulation: A Case Study." In *Proceedings of the Eighteenth Annual Simulation Symposium*, pages 95–107, 1985.

37. Reed, Daniel A. and Allen D. Malony. "Parallel Discrete Event Simulation: The Chandy-Misra Approach." In *Distributed Simulation 1988*, La Jolla CA: SCS, 1988.

38. Reynolds, Paul F., Jr. "A Shared Resource Algorithm for Distributed Simulation." In *Proceedings of the Ninth Annual Int'l Computer Architecture Conference*, pages 259–266, April 1982.

39. Reynolds, Paul F., Jr. "A spectrum of options for parallel simulation." In *Proceedings of the 1988 Winter Simulation Conference*, pages 325–332, December 1988.

40. Shannon, Robert E. *Systems Simulation: The Art and Science*. Englewood Cliffs NJ: Prentice-Hall, 1975.

41. Wieland, Fredrick and others. "Distributed combat simulation and time warp: The model and its performance." In *Distributed Simulation 1989*, La Jolla CA: SCS, 1989.

42. Zhang, Guoqing and Bernard P. Zeigler. "DEVS-Scheme Supported Mapping of Hierarchical Models onto Multiple Processor Systems." In *Distributed Simulation 1989*, La Jolla CA: SCS, 1989.

Vita

Mark Leslie Huson [redacted] After graduating from Springfield High School in 1977, he attended the University of Tulsa (no degree), before enlisting in the Air Force in 1982. He served as an Electronic Warfare Technician at Eglin AFB, Florida, prior to completing his B.S. in Computer Science from the University of Tulsa in 1985 via the Air Force's Bootstrap program. After receiving his commission through OTS in September, 1985, he was stationed at Peterson AFB, Colorado, as a Missile Warning/Space Defense Communications Analyst for HQ AFSPACECOM. In December 1986 he became a Command, Control, Communications Programmer for the Mobile Command and Control System (MCCS), an Ada development project for US Space Command. He completed an M.S. in Systems Management from the University of Southern California in April of 1988, prior to reporting to the Air Force Institute of Technology.

[redacted]

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/89D-10	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, Ohio 45433	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING / SPONSORING ORGANIZATION SDIO Phase I Program Office 8b. OFFICE SYMBOL (If applicable) SDIO/S/PI	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Room 1E149, The Pentagon Washington, D.C. 20301-7100	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
AN EMPIRICAL DEVELOPMENT OF PARALLELIZATION GUIDELINES FOR TIME-DRIVEN SIMULATION

12. PERSONAL AUTHOR(S)
Mark L. Huson, Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM ______ TO ______	1989 December	190

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
12	05		Computerized Simulation, Parallel processing

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Advisor: Dr. Thomas C. Hartrum
Associate Professor
Department of Electrical and Computer Engineering

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. Thomas C. Hartrum	22b TELEPHONE (Include Area Code) (513) 255-3576 22c OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Block 19.

Distributed simulation is an area of research which offers great promise for speeding up simulations. Program parallelization is usually an iterative process requiring several attempts to produce an efficient parallel implementation of a sequential program. This is due to the lack of any standards or guidelines for program parallelization.

In this research effort a Ballistic Missile Defense (BMD) time-driven simulation program, developed by DESE Research and Engineering, was used as a test vehicle for investigating parallelization options for distributed and shared memory architectures. Implementations were developed to address issues of functional versus data program decomposition, computation versus communications overhead, and shared versus distributed memory architectures.

Performance data collected from each implementation was used to develop guidelines for implementing parallel versions of sequential time-driven simulations. These guidelines were based on the relative performance of the various implementations and on general observations made during the course of the research.

END